Curso de React.js AVANZADO

# ¿Qué es React.js?

Es un sistema de plantillas para gestionar las vistas que utiliza un lenguaje de marcado llamado JSX, similar a HTML  
.  
**Características básicas de react**  
.

* Está basado en **componentes**: Toda la UI se divide en elementos más pequeños llamados componentes, en react, todo es un componente,
* Es [**declarativo**](https://dev.to/itsjzt/declarative-programming--react-3bh2): **expresa la lógica de un cálculo sin describir su flujo de control**, es un estilo de programación en el que el que se define “qué” es la solución sin importar cómo se llegó a ella.

# Proyecto y tecnologías que usaremos

En este curso realizaremos una aplicación muy parecida a Instagram, llamada petgram. Tendremos nuestras rutas, gestión de usuarios y likes.

Utilizaremos como empaquetador y transpilador:

* [Webpack](https://platzi.com/cursos/webpack/)
* [Babel](https://babeljs.io/)

Estilado con CSS en JS con:

* [styled-components](https://www.styled-components.com/)

Como linter utilizaremos:

* [Standard JS](https://standardjs.com/)

Para fetching (obtención) de datos:

* [GraphQL](https://platzi.com/cursos/graphql/)
* [React Apollo](https://www.apollographql.com/docs/react/)

Para el enrutado de la SPA utilizaremos:

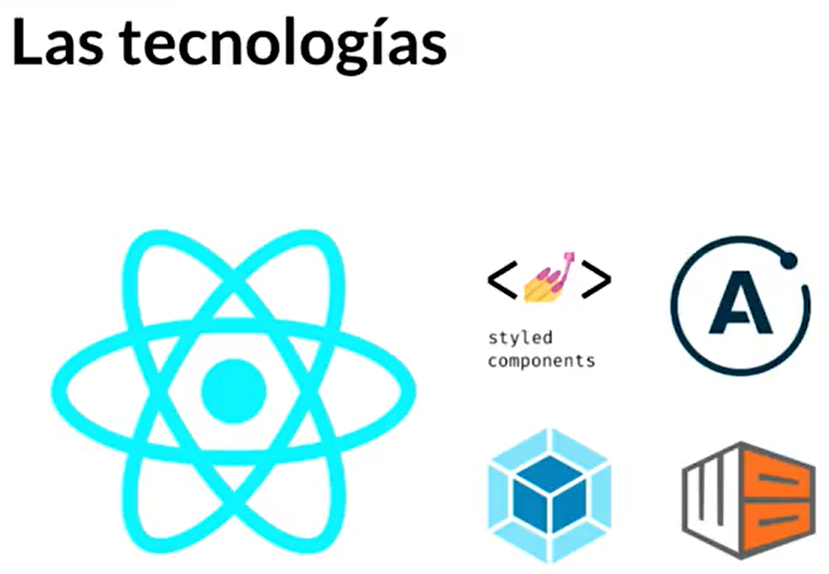
* [Reach Router](https://reach.tech/router)

Para las buenas prácticas utilizaremos:

* [Lighthouse](https://developers.google.com/web/tools/lighthouse/?hl=es)
* [Cypress](https://www.cypress.io/)

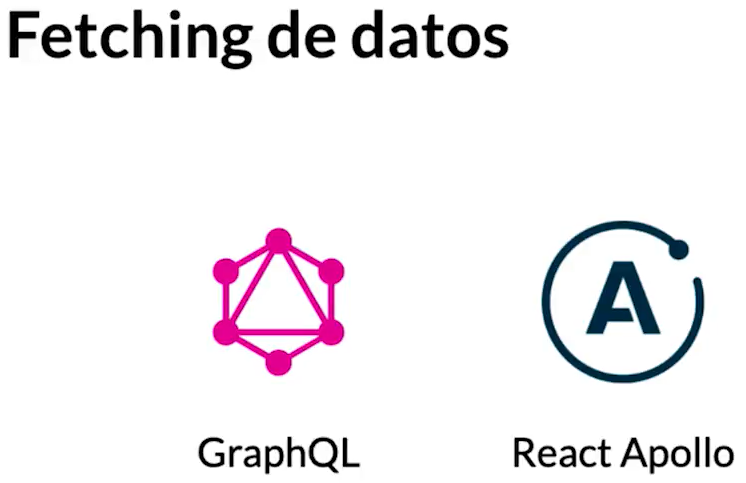
Por último haremos SEO, PWA y Deploy con:

* [React Helmet](https://github.com/nfl/react-helmet)
* [Workbox](https://developers.google.com/web/tools/workbox/)
* [Progressive Web App](https://platzi.com/cursos/pwa-react-js/)
* [Deply con Vercel](https://platzi.com/cursos/deploy-now/)



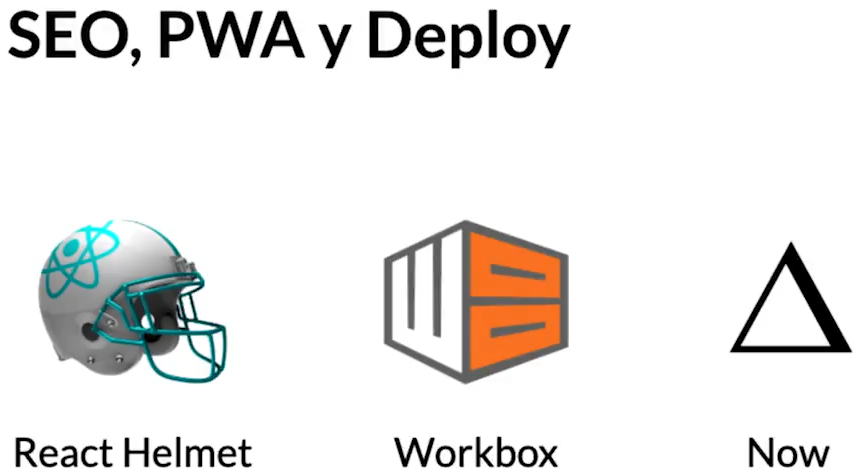












# Clonando el repositorio e instalando Webpack

Pasos para iniciar nuestro proyecto:

* **Paso 1:** Vamos a clonar el repositorio desde [github.com/midudev/curso-platzi-react-avanzado](https://github.com/midudev/curso-platzi-react-avanzado) usando git clone URL\_DEL\_REPO en nuestra consola.
* **Paso 2:** Vamos a instalar webpack y webpack-cli como dependencias de desarrollo con: npm i webpack wepack-cli --save-dev.
* **Paso 3:** Crearemos una carpeta llama src y dentro de ella un archivo index.js en el cual colocaremos solo un console.log('Empezamos el curso!').
* **Paso 4:** Crearemos en el root de nuestro proyecto un archivo webpack.config.js el cual tendrá toda la configuración de webpack
* **Paso 5:** Instalaremos html-webpack-plugin con: npm i html-webpack-plugin --save-dev.
* **Paso 6:** Instalaremos webpack-dev-server con: npm i webpack-dev-server --save-dev.
* **Paso 7:** Añadiremos un nuevo script llamado dev: "dev": "webpack-dev-server".

// webpack.config.js

**const** HtmlWebpackPlugin = require("html-webpack-plugin")

module.exports = {

output: {

filename: 'app.bundle.js'

},

plugins: [

**new** HtmlWebpackPlugin()

]

}

Nota:

Comando que corregir

npx webpack --entry ./src/index.js

"dev": "webpack serve --open",

# Instalación de React y Babel

En esta clase vamos a configurar React instalando las dependencias npm i react react-dom y Babel para poder transpilar código *jsx* a JavaScript Vanilla con: npm i @babel/core @babel/preset-env babel-loader @babel/preset-react --save-dev.

Ahora añadiremos en nuestro webpack.config.js lo siguiente:

// webpack.config.js

{/\*...\*/}

module.exports = {

{/\*...\*/}

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader',

options: {

presets: [

'@babel/preset-env',

'@babel/preset-react'

]

}

}

}

]

}

}

# Linter, extensiones y deploy con Vercel

En esta clase haremos que el desarrollo sea más ágil y correcto siguiendo los siguientes pasos:

* Vamos a instalar StandardJS como dependencia de desarrollo con: npm i standard --save-dev. StandardJS nos va a servir de Linter para una mejor escritura de JavaScript/React.
* Agregaremos un nuevo script en nuestro package.json: ""lint"": ""standard"".
* Ahora vamos a ignorar aquellos archivos que no queremos que el Linter arregle, añadiremos en nuestro package.json lo siguiente:

""standard"": [

""ignore"": [

""/api/\*\*""

]

]

* Ahora, queremos que nuestro Linter nos detecte los errores a medida que vamos escribiendo, para hacer esto añadimos lo siguiente a nuestro package.json:

""eslintConfig"": {

""extends"": [

""./node\_modules/standard/eslintrc.json""

]

}

* Ahora debemos tener lo siguiente en nuestro editor de código para que funcione todo al pie de la letra:
  + Tener instalada la extensión [ESLint](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint)
  + Si quieres que al guardar los cambios se formatee tu código deberás tener instalada la extensión [Prettier](https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode)
  + Tener las siguientes configuraciones en VSCode:
    - Format On Save: false
    - Prettier: Eslint Integration: true
    - Eslint: Auto Fix On Save: true
* Ahora utilizaremos [Vercel](https://zeit.co/login) para hacer el deploy de nuestro proyecto.
* Descargaremos e instalaremos [Vercel](https://zeit.co/download) para que nos registre de una manera mucho más fácil los tokens de acceso y podamos continuar con el curso.
* Entraremos a la carpeta de api y notaremos que ya tiene un archivo Vercel.json que preparamos para ti con toda la configuración necesaria para hacer el deploy.
* Para desplegar el proyecto del backend haremos lo siguiente en nuestra terminar:
  + cd api
  + Cambiamos el name de la aplicación en el Vercel.json
  + Finalmente ejecutamos Vercel
* Ahora para desplegar nuestro front haremos lo siguiente:
  + Crearemos un archivo Vercel.json en el root de nuestro proyecto con lo siguiente:

{

""version"": 2,

""name"": ""petgram"",

""builds"": [

{

""use"": ""@Vercel/static-build"",

""src"": ""package.json""

}

],

""routes"": [

{

""src"": ""(.\*).js"",

""dest"": ""/$1.js""

},

{

""src"": ""(.\*).json"",

""dest"": ""/$1.json""

},

{

""src"": ""/.\*"",

""dest"": ""index.html""

}

]

}

* En nuestro package.json añadiremos el siguiente script: ""Vercel-build"": ""npm run build"".
* Finalmente en la raíz de nuestro proyecto ejecutaremos Vercel para que nos dé una URL en la que se verá nuestro proyecto."

Seria bueno que actualizarán este video. Now ya no es now ahora es vercel y tiene algunas diferencias, puede llegar a ser confuso. En todo caso lo que deben hacer es:

1. Instalar vercel a traves del comando **npm i -g vercel**
2. Despues de instalar vercel deben hacer **vercel login** y verificar su correo electronico, para que la instancia de versel local este conectada con el remoto y de esta forma evitar los tokens.
3. Luego si realizar los pasos que dice el profe, pero con la difencia que el comando **no es now, es vercel**

Si alguno tuvo inconvenientes en VSCode con la siguiente validación

* Format On Save: false
* Prettier: Eslint Integration: true
* Eslint: Auto Fix On Save: true

Instalé las dependencias de desarrollo

npm i prettier eslint-config-prettier eslint-plugin-prettier -D

Y configure manualmente en Open Settings (settings.json) ubicado en la parte superior derecha del editor (Tiene forma de {} o como un archivo" y revisan si su archivo de configuración tiene las siguiente lineas y sino las insertan

"editor.codeActionsOnSave": {

"source.fixAll.eslint": **true**

}

"eslint.format.enable": **true**,

"prettier.eslintIntegration": **true**,

eslint.autofixonsave: true esta obsoleto por eso debe reemplazarse con editor.codeActionsOnSave

# ¿Qué es CSS-in-JS?

Antes la forma en la que construíamos nuestras aplicaciones era con HTML-centric:

<button className='button button-red'>

Click here!

button>

**const** button = document.querySelector('button')

button.addEventListener('click', **function** () {

doSomething()

})

.button {

border-radius: 4px;

}

.button-red {

background: red;

color: #fff;

}

Luego pasamos a JavaScript-centric:

render () {

**return** (

<button

className='button button-red'

onClick={doSomething}

>

Click here!

button>

)

}

.button {

border-radius: 4px;

}

.button-red {

background: red;

color: #fff;

}

Ahora hoy en día pasamos a CSS-in-JS nos permite no solo hacer el jsx si no colocar los estilos en el mismo JavaScript, de la siguiente manera:

**const** Button = styled.button`

border-radius: 4px;

${props => props.accent && `

background: red;

color: #fff;

`}

render () {

return (

Click here!

)

}

Con CSS-in-JS no debemos preocuparnos por tener dos clases igual, no nos debemos preocupar por el Critical Path CSS, tendremos un CSS mucho más óptimo.

En esta clase utilizaremos [styled-components](https://www.styled-components.com/) el cual es una biblioteca para CSS-in-JS pensada más que todo para React

## Styled-components

Los styled-components son una de las nuevas formas de usar CSS en JavaScript moderno. Está basado en los CSS modules, una forma de escribir CSS que se enfoca el que el alcance de los estilos funcione por componente y no se filtren a ningún otro elemento de la página.  
.  
Los styled components se basan en crear componentes de carácter estético para hacer código más semántico y evitar los problemas de especificidad que trae el CSS tradicional.  
.  
Los styled components le permiten escribir CSS simple en sus componentes sin preocuparte por las colisiones de nombres de clase y haciendo uso del poder de JavaScript para manejar estilos con lógica (basado en props).  
.  
**Nota: en styled components escribes CSS exacto, así que prácticamente todo lo que puedes hacer en CSS lo puedes hacer en styled components y se escribe igual.**  
.  
Es bastante sencillo, sólo tenemos que seguir 3 pasos:

1. importar styled components

**import** styled **from** 'styled-components'

1. crear un componente con estilo basado en algún elemento html

**const** DivConEstilo = styled.div`

color:red;

background:black;

`

1. Utilizar el componente como si fuera cualquier componente de React

**const** Contenedor = () =>{

**return** (

<DivConEstilo>

Hola Styled Component

</DivConEstilo>

)

}

<https://styled-components.com/>

# Creando nuestro primer componente: Category

npm install styled-components

Vamos a construir nuestro primer componente el cual será y usaremos [styled-components](https://www.styled-components.com/) para hacer los estilos de nuestro proyecto.

Para que el código del styled-component esté mejor formateado en nuestro editor utilizaremos esta extensión [vscode-styled-components](https://marketplace.visualstudio.com/items?itemName=jpoissonnier.vscode-styled-components).

Una buena practica seria definir desde styled-components, los globalStyles, de esta manera podras definir los valores que mejor te convengan… aqui un ejemplo:

**import** { createGlobalStyle } **from** 'styled-components';

**const** GlobalStyles = createGlobalStyle`

html {

box-sizing: border-box;

font-size: 10px;

}

\*, \*:before, \*:after {

box-sizing: inherit

}

body {

padding: 0;

margin: 0;

font-size: 1.5rem;

font-family: sans-serif;

}

h1, h2, h3, h4, h5, p {

margin: 0;

}

a {

text-decoration: none;

color: #393939

}`;

**export** **default** GlobalStyles;

# Creando ListOfCategories y estilos globales

Para crear nuestros estilos globales con styled-components haremos lo siguiente:

// src/GlobalStyles.js

**import** { createGlobalStyle } **from** 'styled-components'

**export** **const** GlobalStyle = createGlobalStyle`

html {

box-sizing: border-box;

font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen, Ubuntu, Cantarell, 'Open Sans', 'Helvetica Neue', sans-serif;

}

\*, \*::before, \*::after {

box-sizing: inherit;

}

ul, li, h1, h2, h3, p, button {

margin: 0;

}

ul {

list-style: none;

}

button {

background: transparent;

border: 0;

outline: 0;

}

body {

background: #fefefe;

height: 100vh;

margin: 0 auto;

max-width: 500px;

overscroll-behavior: none;

width: 100%;

}

#app {

box-shadow: 0 0 10px rgba(0, 0, 0, 0.05);

overflow-x: hidden;

min-height: 100vh;

padding-bottom: 10px;

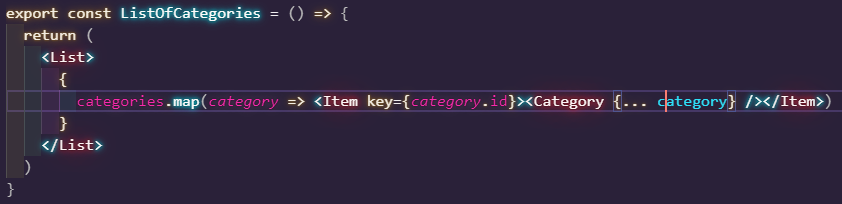
}`

# Usar informacion real de las categorias

Muy top pasar los props con el rest operator… , recibirlos como parametros directamente en el componente y que mas nos deparará este curso 😄… chavales… pillad café y arándanos que esto se pone bueno.

Por si alguien tiene dudas o simplemente para recordar conceptos🧐:

* [Método map()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map)
* [Operador rest / spread](https://codingwithspike.wordpress.com/2016/06/11/javascript-rest-spread-operators/)



The spread operator can be used to turn an array of values onto a set of function parameters. Returning to our add() example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | function add(...numbersToAdd) { // This is a Rest parameter    return numbersToAdd.reduce((sum, next) => sum + next);  }    var numbers = [1, 2, 3];  add(...numbers); // this is a Spread operator  // The above is functionally the same as:  add(1, 2, 3); |

This is extremely useful when you have an unknown (at dev time) number of arguments that you want to pass to a function. You can .push() several values into an array then use them as parameters to a function.

# Creando PhotoCard y usando react-icon

En esta clase vamos a construir nuestro componente que tendrá las imágenes de nuestro timeline llamado .

Usaremos [react-icons](https://react-icons.netlify.com/#/) para darle una propuesta visual mucho más amigable a nuestro proyecto con íconos como Font Awesome, Ionicons, Material Design Icons y mucho más que podremos usar.

npm install react-icons

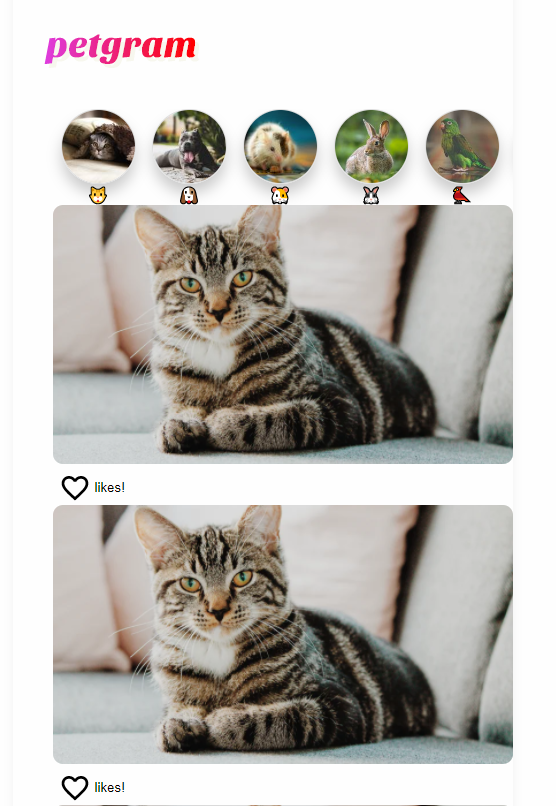




# SVGR: de SVG a componente de ReactJS

En esta clase usaremos [maketext.io](https://maketext.io/) para crear nuestro logo y descargarlo en SVG. Posteriormente a esto utilizaremos [SVGOMG](https://jakearchibald.github.io/svgomg/) para optimizar nuestro logo y tener una mejor versión para usarla en nuestro proyecto.

Luego para convertir nuestro logo *svg* en un componente utilizaremos [SVGR](https://www.smooth-code.com/open-source/svgr/playground/).

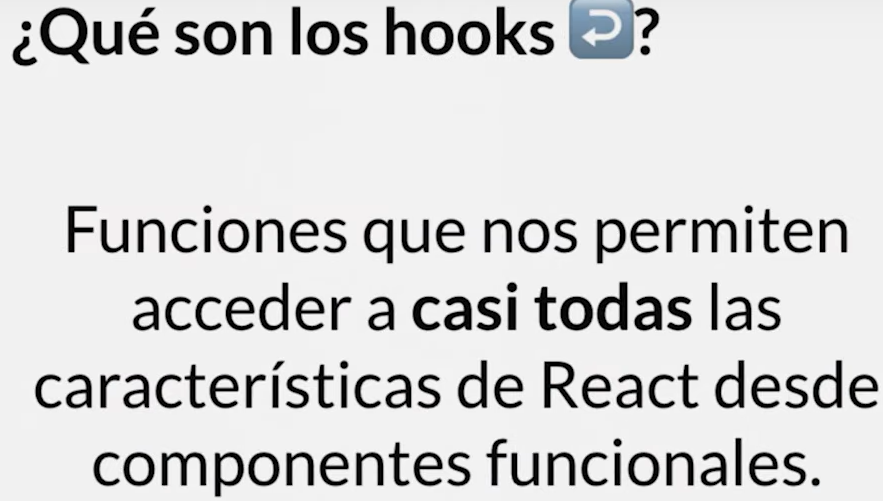
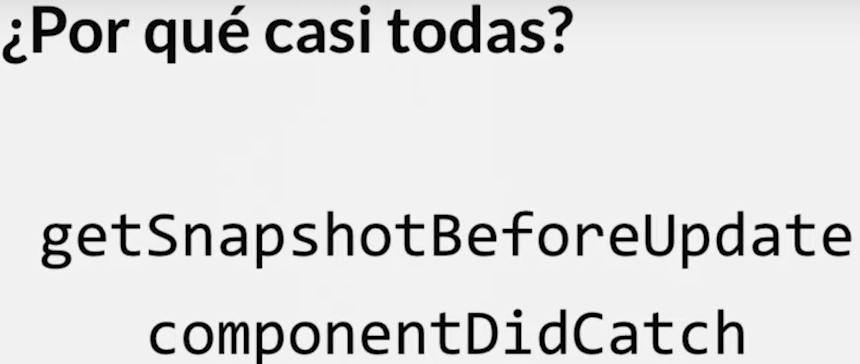


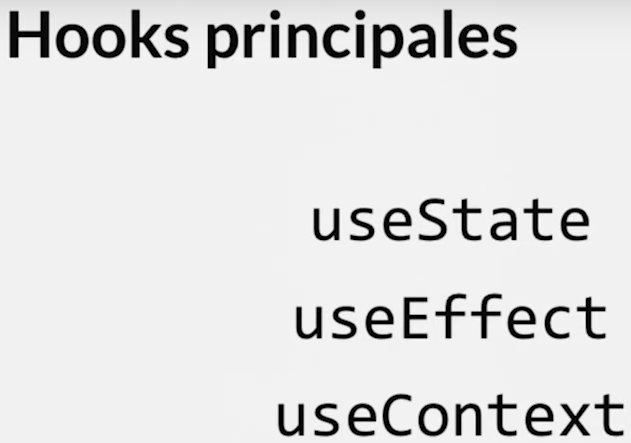
# Creando animaciones con keyframes



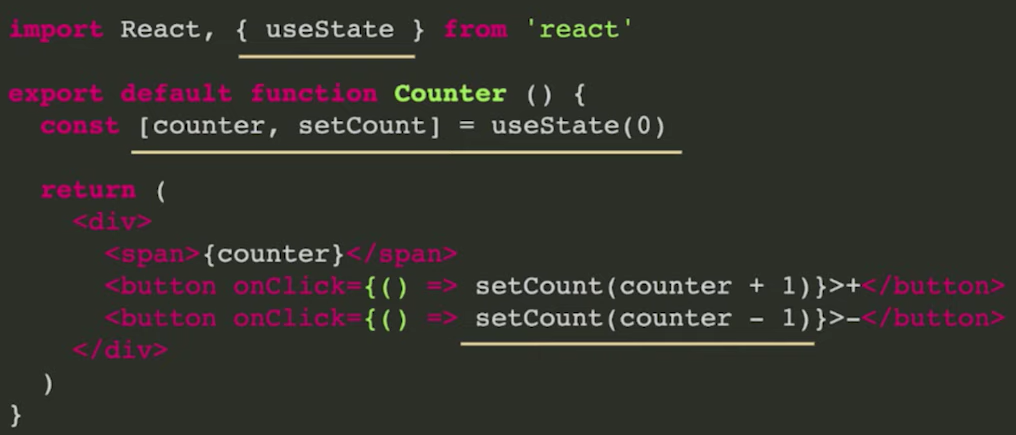


# ¿Qué son los Hooks?

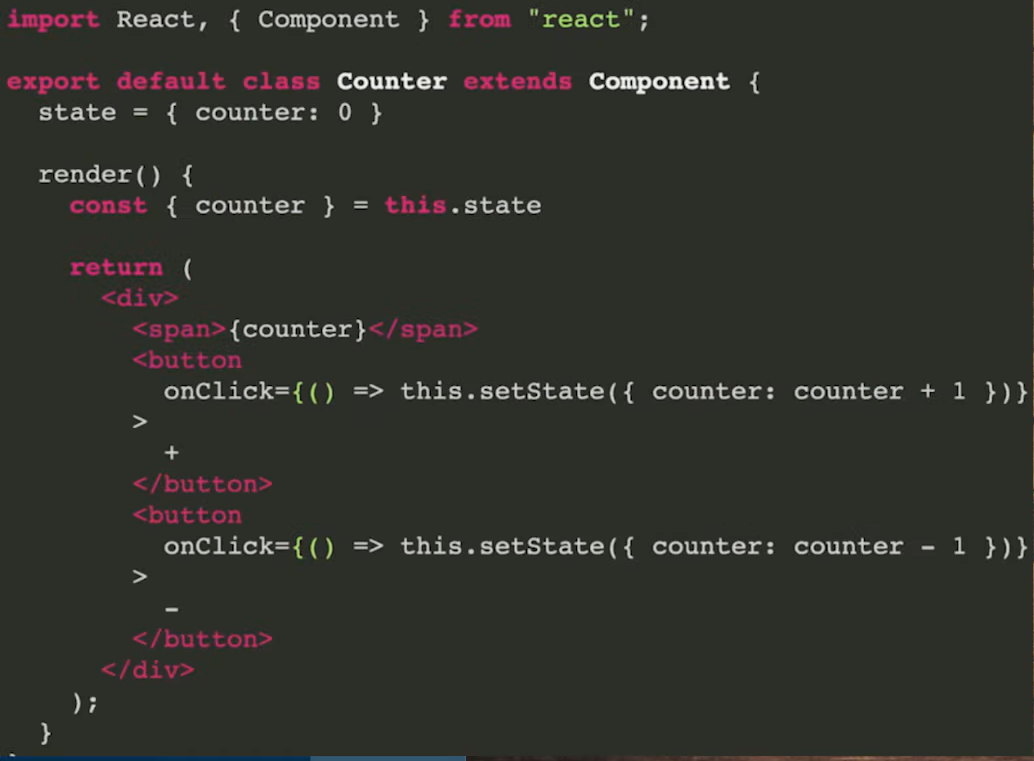
 

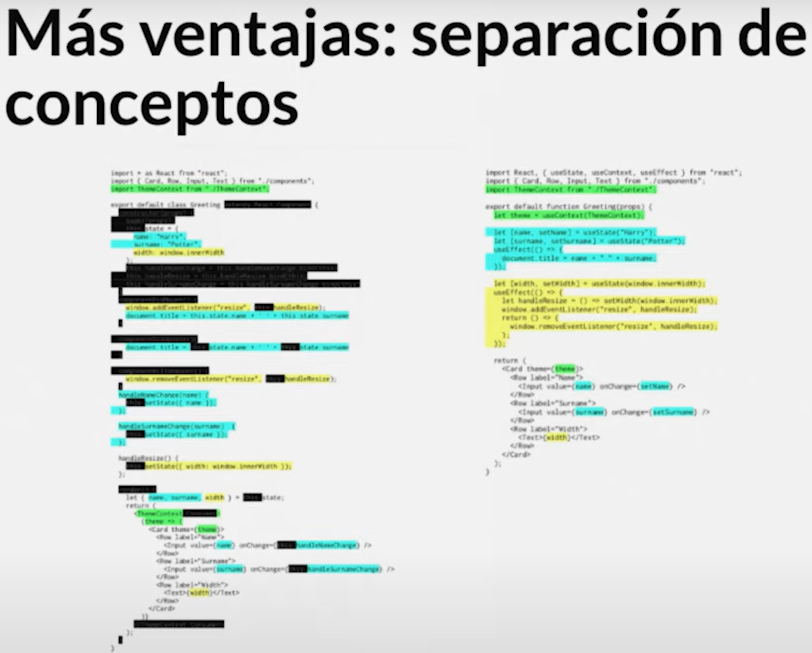
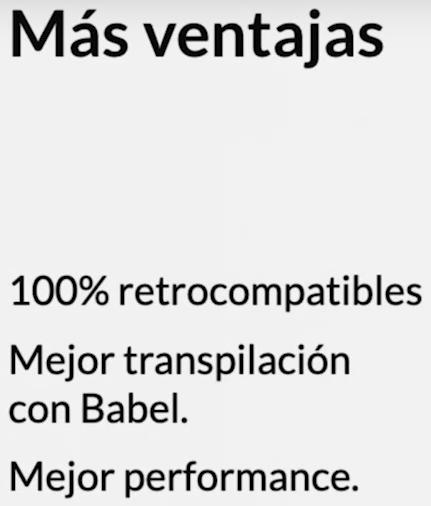
Con Hooks



Sin Hooks



**¿Qué son los Hooks?**  
Funciones que nos permiten acceder a casi todas las características de react desde componentes funcionales.  
Las características que por ahora no se pueden acceder son:  
getSnaphotBeforeUpdate  
componentDidCatch  
🔧  
Los hooks principales son  
- useState: para añadir un estado local en el componente  
- useEffect: permite ejecutar una funcion cada vez que rendericemos nuestro componente  
- useContext: permite acceder a la context API para obtener valores que se utilizaran en toda la aplicacion de forma global, sin necesidad de pasarla por las props.

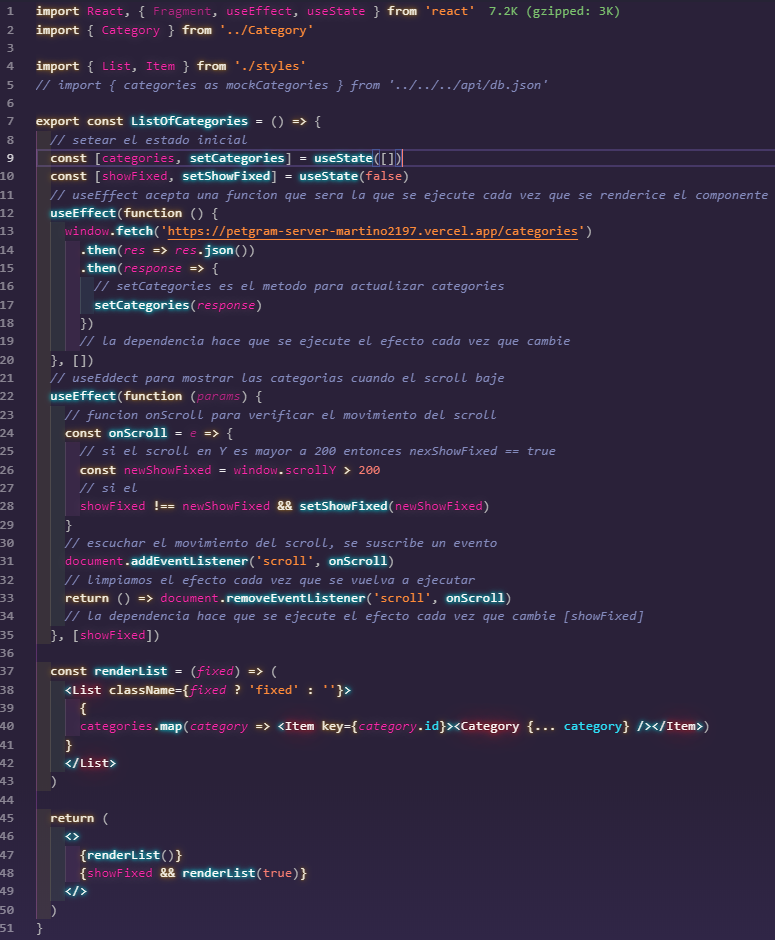
 

# useEffect: limpiando eventos

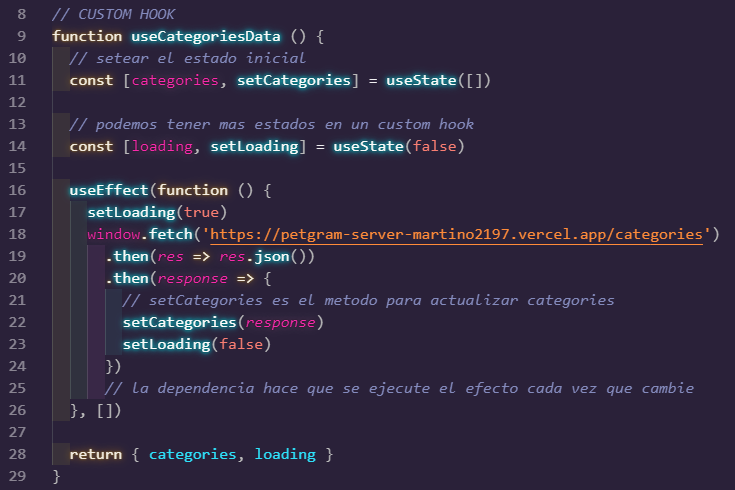
## useEffect: accediendo al ciclo 🌀 de vida de nuestro componente

El ciclo de vida de los componentes en React permitía en nuestros componentes con class poder ejecutar código en diferentes **fases de montaje, actualización y desmontaje.** De esta forma, podíamos añadir cierta funcionalidad en las distintas etapas de nuestro componente.

Con los hooks también podremos acceder a ese ciclo de vida en nuestros componentes funcionales aunque de una forma más clara y sencilla. Para ello usaremos **useEffect, un hook que recibe como parámetro una función que se ejecutará cada vez que nuestro componente se renderice, ya sea por un cambio de estado, por recibir props nuevas o, y esto es importante, porque es la primera vez que se monta.**



# useCategoriesData





category/index.js

**import** React **from** 'react';

**import** { ContainerCategorySkeleton, CategoryImage, CategoryTitle } **from** './styles';

**export** **const** CategorySkeleton = props => {

console.log(props)

**return** (

<ContainerCategorySkeleton>

<CategoryImage light={props.light} />

<CategoryTitle light={props.light} />

</ContainerCategorySkeleton>

)

}

category/style.js

import styled, { css } from 'styled-components';

import { skeletonStyle } from '../../../styles/skeleton';

export const ContainerCategorySkeleton = styled.**div`**

display: flex;

flex-**direction:** column;

align-items: center;

`;

export const CategoryImage = styled.**div`**

width: 75px;

height: 75px;

**border-radius:** 50%;

${

props => css`

${skeletonStyle(props.light)}

`

}

`;

export const CategoryTitle = styled.**div`**

width: 26px;

height: 15px;

margin-top: 8px;

${

props => css`

${skeletonStyle(props.light)}

`

}

`;

/animation/skeleton/style.js (funciona para cualquier container)

import { css, keyframes } from 'styled-components';

const skeletonBackground = (light) => (

css`

background: ${ !light

? css`linear-gradient(-90deg, #C1C1C1 0%, #F8F8F8 50%, #C1C1C1 100%)`

: css`linear-gradient(-90deg, #F0F0F0 0%, #F8F8F8 50%, #F0F0F0 100%)`};

background-size: 400% 400%;

animation: ${skeletonLoading} 1.2s ease-in-out infinite;

`

)

const skeletonLoading = keyframes`

from {

background-position: 0% 0%;

}

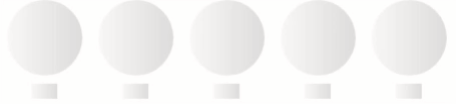
to {

background-position: -135% 0%;

}

`;

export const skeletonStyle = (light = **true**) => skeletonBackground(light);

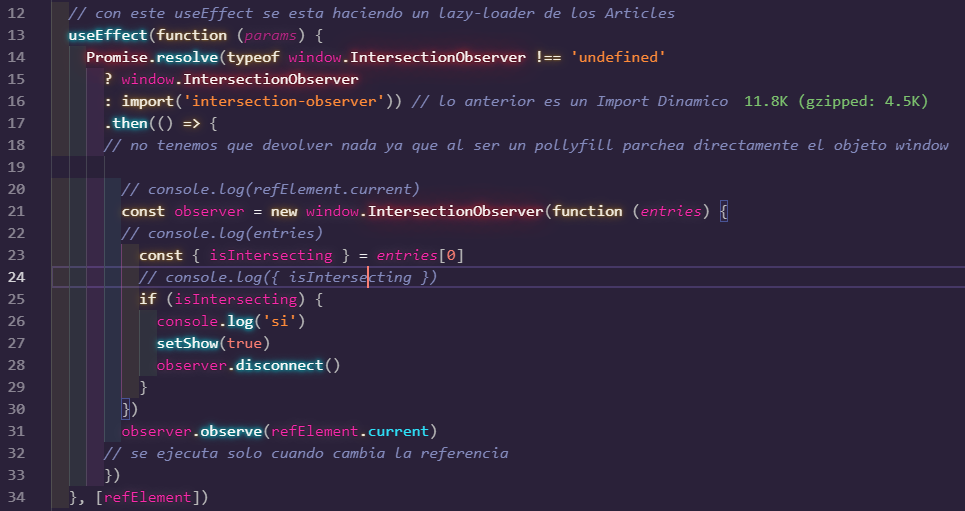
Resultado final:  


# Usando Intersection Observer



# Uso de polyfill de Intersection Observer e imports dinámicos

En esta clase añadiremos soporte a navegadores viejos con un [polyfill de Intersection Observer](https://github.com/w3c/IntersectionObserver/tree/master/polyfill).



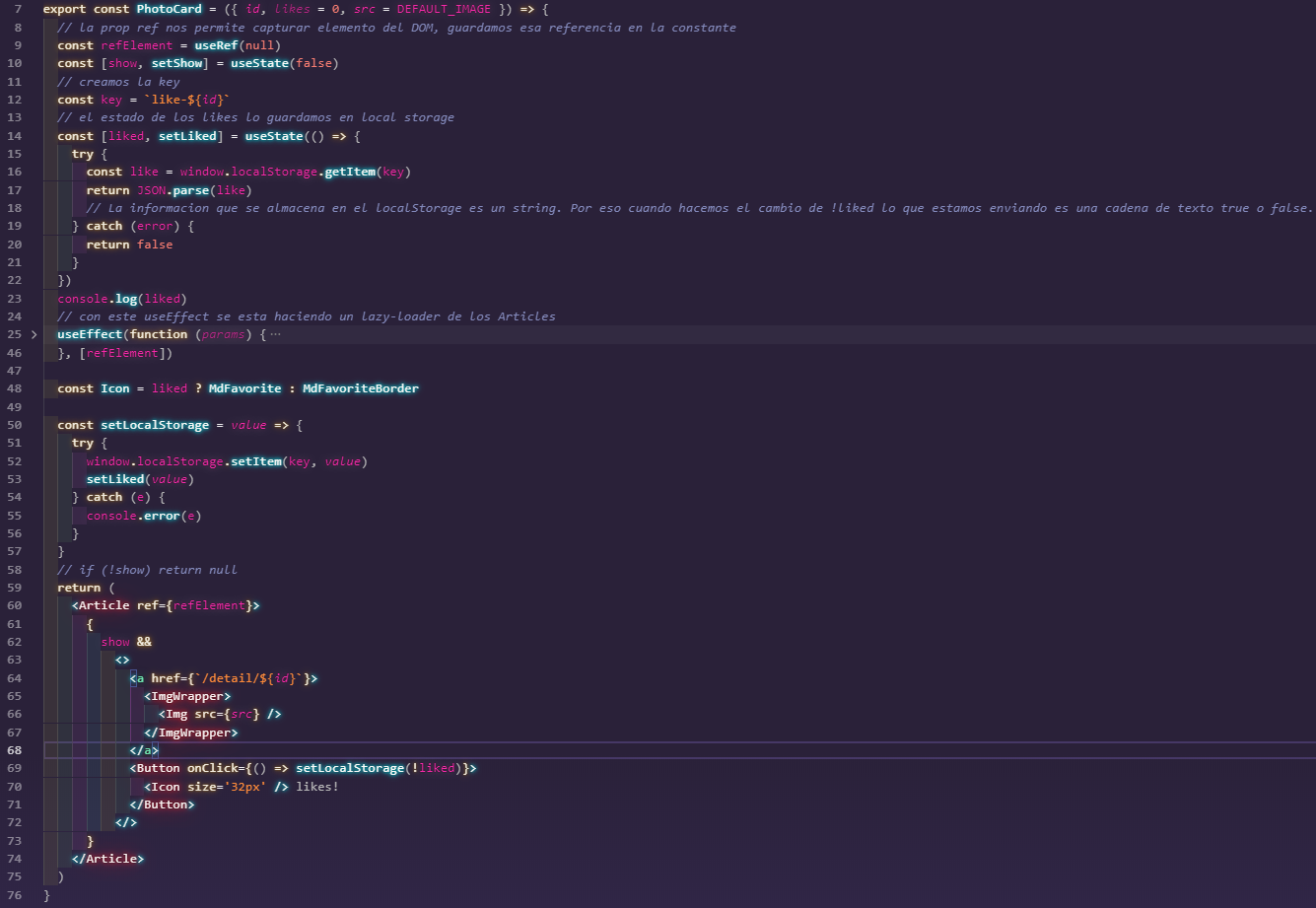
Otra manera de preguntar por una key en un objeto puede ser:

'IntersectionObserver' **in** window

Eso te devolverá un boolean.

Recientemente salió ECMA 2020 y ya se pueden usar imports dinámicos tranquilamente de manera natural, sin necesidad de nada extra.

# Usando el localStorage para guardar los likes



Le falto el JSON.parse(like) cuando obtiene el valor del local storage, de lo contrario siempre que le das like a una foto no volvera a false, porque lo que retorna es un string con el valor “false” hay que parsearlo para que lo lea, al principio funciona ya que los valores que devuelven son null! (osea falsos)

**const** [liked, setLiked] = useState(() => {

**try** {

**const** like = window.localStorage.getItem(key)

**return** JSON.parse(like)

} **catch** (e) {

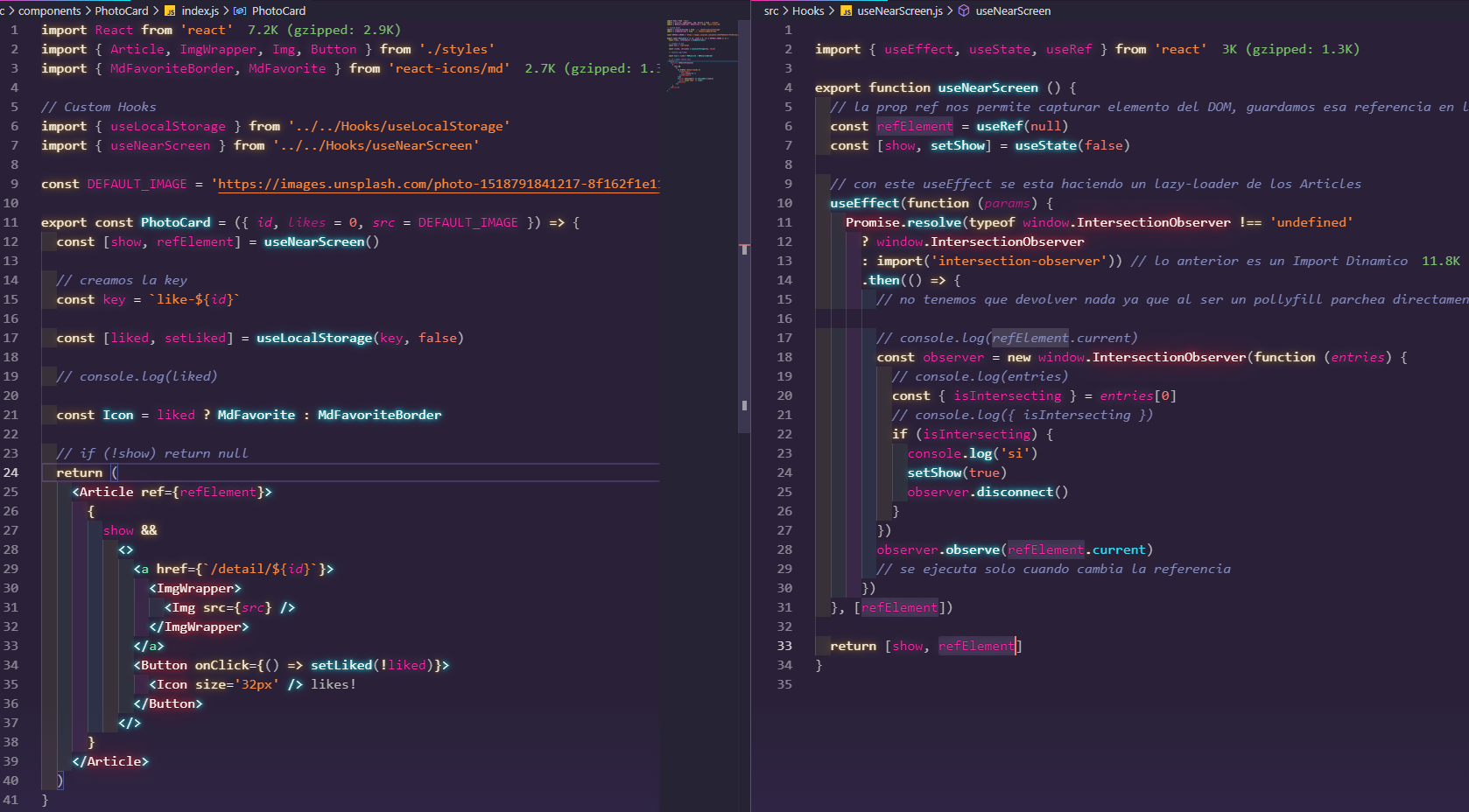
console.log('error!!', e)

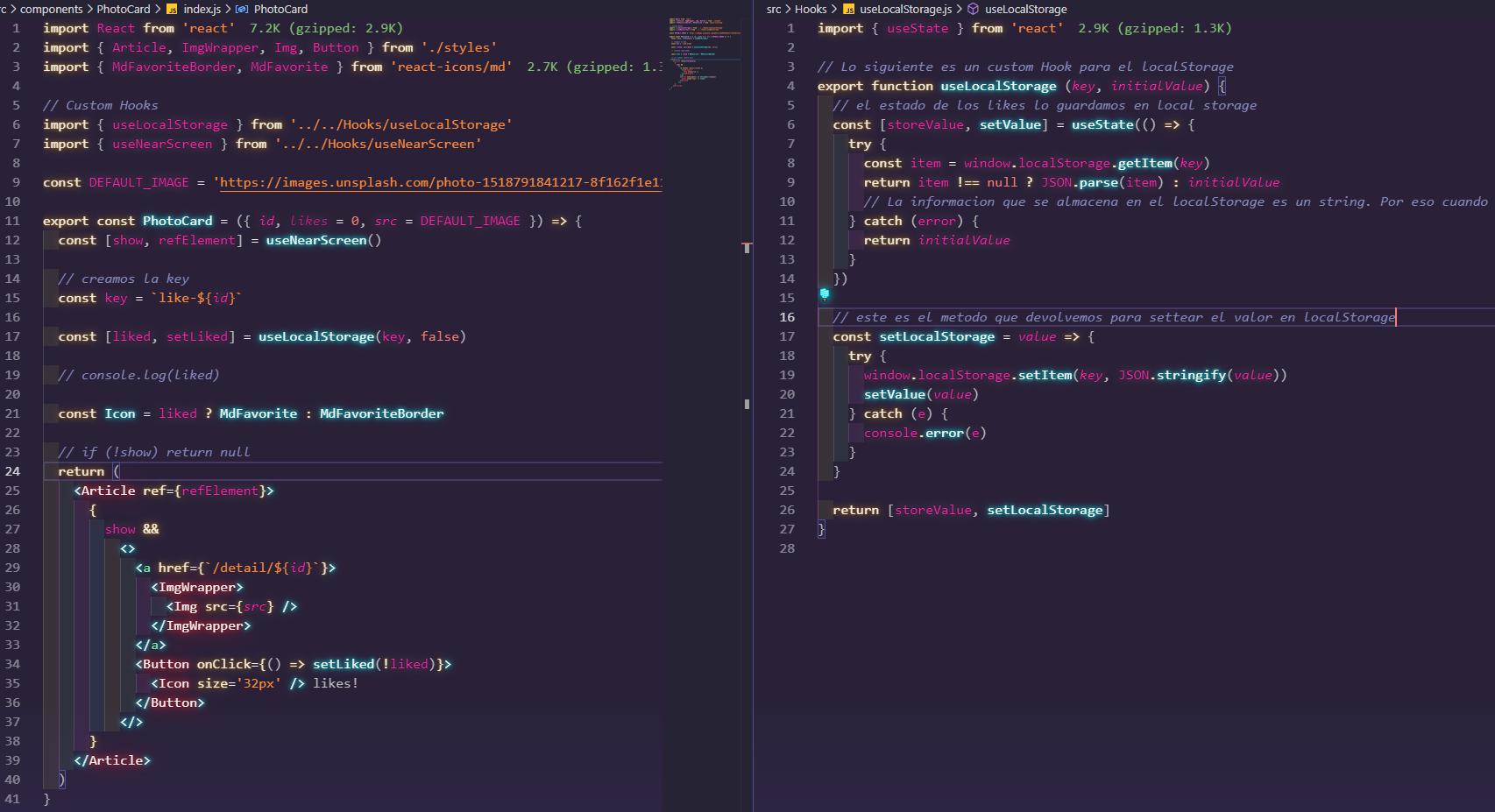
}

})

Cierto lo habia olvidado. La informacion que se almacena en el localStorage es un string. Por eso cuando hacemos el cambio de !liked lo que estamos enviando es una cadena de texto true o false. Si aun asi tienen duda pueden comprobarlo con typeof like

# Custom Hooks: useNearScreen y useLocalStorage





Cree este custm hook ara hacer fetch desde cualquier URL

**import** { useState, useEffect } from 'react'

**export** **const** useFetch = url => {

**const** [data, setData] = useState([])

**const** [loading, setLoading] = useState(**false**)

useEffect(() => {

setLoading(**true**)

**const** fetchdata = **async** () => {

**const** response = **await** window.fetch(url)

**const** data = **await** response.json()

setData(data)

setLoading(**false**)

}

fetchdata()

}, [])

**return** { data: data, loading }

}

Lo use en LisCategories  
y solo lo cambie asi

**const** { data: categories, loading } = useFetch('https://petgram-server.javierfuentesm.now.sh/categories')

# ¿Qué es GraphQL y React Apollo? Inicializando React Apollo Client y primer HoC

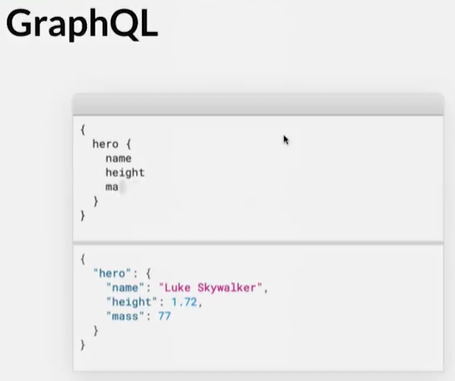
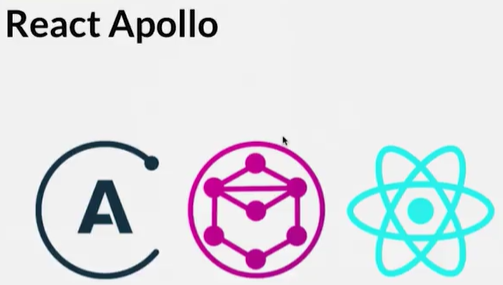
 

Gráfico El es un lenguaje creado por Facebook que nos permite recuperar los datos que justamente necesitamos en nuestra aplicación Los primeros que nos permite describir nuestros datos, de forma que podemos decir cada campo qué tipo tiene y de esta forma es Tags validaciones ocurrirán en la parte tanto del cliente como del servidor, Además.

Podemos recuperar justo la información que necesitamos, así que optimizará los recursos de nuestra red y también vamos a tener resultados que son predecibles porque nosotros mismos vamos a identificar qué es lo que necesitamos utilizar en nuestra aplicación.

GraphQL es un lenguaje creado por Facebook para obtener solo los datos que necesitamos en nuestra aplicación. Imaginemos que tenemos una REST API con WordPress, él nos entrega un JSON con mucha *data* innecesaria y solo queremos unos cuantos *fields*; GraphQL hace posible obtener únicamente esa data.

React Apollo es un cliente que nos va a permitir conectarnos a un servidor GraphQL.

Vamos a instalar npm i apollo-boost, una utilidad que nos permite inicializar nuestra conexión con un servidor de Apollo muy rápidamente y sin configuración, necesitaremos npm i react-apollo el cual es la integración de Apollo con React y finalmente npm i graphql.

Configuraremos nuestro index.js para inicializar nuestro cliente de Apollo con GraphQL de la siguiente manera:

// index.js

{/\*...\*/}

**import** ApolloClient **from** 'apollo-boost'

**import** { ApolloProvider } **from** 'react-apollo'

{/\*...\*/}

**const** client = **new** ApolloClient({

uri: 'URL\_DEL\_BACKEND' // En el proyecto usamos https://petgram-server.midudev.now.sh/graphql aquí debes colocar el tuyo

})

ReactDOM.render(

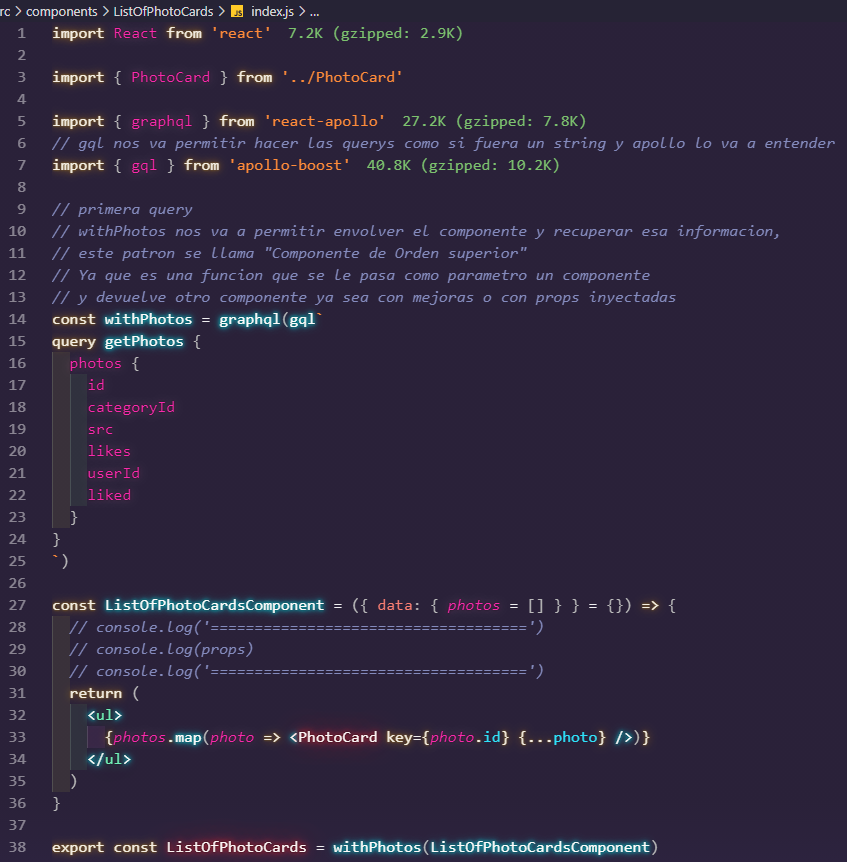
<ApolloProvider client={client}>

<App />

ApolloProvider>,

document.getElementById('app')

)



npm install apollo-boost react-apollo graphql

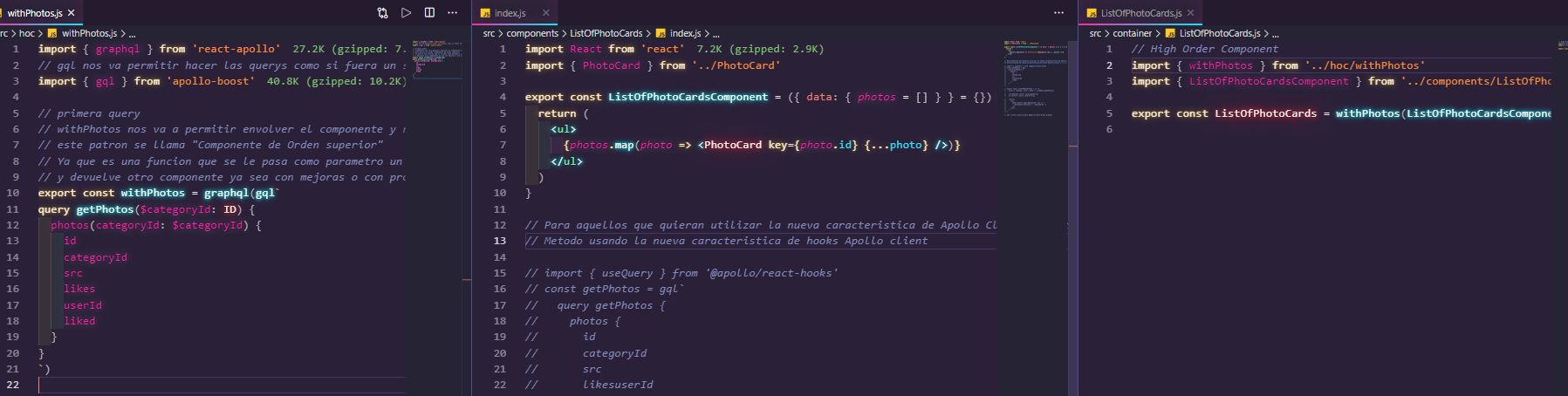
para ocupar hooks Tambien se tiene que agregar npm install @apollo/react-hooks

# Parámetros para un query con GraphQL

¿Podríamos decir que:

* M: withPhotos
* V: ListOfPhotoCardComponent
* C: ListOfPhotoCard

??? es un pequeño MVC del lado del cliente?



Estoy realizando este ejercicio utilizando hooks, @apollo/client como cliente y proveedor

En el entrypoint de la app importamos el ApolloClient, ApolloProvider de @apollo/client

**/index.js**

**import** React **from** 'react'

**import** ReactDOM **from** 'react-dom'

**import** { App } **from** './App'

**import** { ApolloClient, InMemoryCache, ApolloProvider } **from** '@apollo/client'

**const** client = **new** ApolloClient({

uri: process.env.API\_URL,

cache: **new** InMemoryCache()

})

ReactDOM.render(

<ApolloProvider client={client}>

<App />

</ApolloProvider>

, document.getElementById('app'))

El query lo separé en un custom hook

**import** { gql, useQuery } from '@apollo/client'

export **function** useGetPhotos (categoryId) {

const GET\_PHOTOS = gql`

query getPhotos($categoryId: ID) {

photos (categoryId: $categoryId){

id

categoryId

src

likes

liked

userId

}

}`

const { loading, error, data } = useQuery(GET\_PHOTOS, { variables: { categoryId } })

return { loading, error, data }

}

y ListOfPhotocards.js implementa el custom hook de la forma

**import** React **from** 'react'

**import** { PhotoCard } **from** '../PhotoCard'

**import** { useGetPhotos } **from** '../../hooks/useGetPhotos'

**export** **const** ListOfPhotoCards = ({ categoryId }) => {

**const** { loading, error, data } = useGetPhotos(categoryId)

**if** (loading) **return** <p>Loading...</p>

**if** (error) **return** <p>error...</p>

**return** (

<ul>

{data.photos.map((photo) => (

<PhotoCard key={photo.id} id={photo.id} {...photo} />

))}

</ul>

)

}

Mas información en mi [repo](https://github.com/cosmosoftroot/petgram)

# Usar render Props para recuperar una foto

Me costó un poco entender lo de las render props, pero después de investigación externa, es relativamente fácil de entender.  
.  
La técnica de **render props** está basada en la propiedad **children** de los componentes de react, envés de renderizar un componente hijo en el componente padre, **renderizas una función que devuelve un componente**, a la función le pasas como **parámetros los objetos que desees** (puede ser un fetching de base de datos como lo vimos en esta clase) y puedes **acceder a la información** como si ya la tuvieras a la hora de **montar el componente padre con un hijo** de una **función**.

## [Un ejemplo](https://github.com/behagoras/render-props-experiment.git)

.  
Primero creamos un componente que haga toda la lógica del fetching de datos y regrese una función:

const GetPokemon = ({ children, pokemonNumber }) => {

const [pokemon, setPokemon] = useState([]);

useEffect(() => {

window.fetch(`https://pokeapi.co/api/v2/pokemon/${pokemonNumber}/`)

.then((res) => res.json())

.then((myPokemon) => {

setPokemon({

src: myPokemon.sprites.front\_default,

name: myPokemon.species.name,

id: myPokemon.id,

});

})

.catch((error) => { console.error(error); });

}, []);

return ( // acá toda hacer un return del children como función

<div>

{children(

{ pokemon }, // en los argumentos dejamos la información que queramos que sea accesible para otros componentes

)}

</div>

);

};

.  
Después ya podemos acceder a la data del componente al invocarlo y pasar en el children una función con el mismo argumento que fue declarada:  
.

<GetPokemon

pokemonNumber={key + 1}

key={key + 1}

>

{

({ pokemon = {} }) => (

<Pokemon

src={pokemon.src}

pokemonName={pokemon.name}

pokemonNumber={pokemon.id}

key={key + 1}

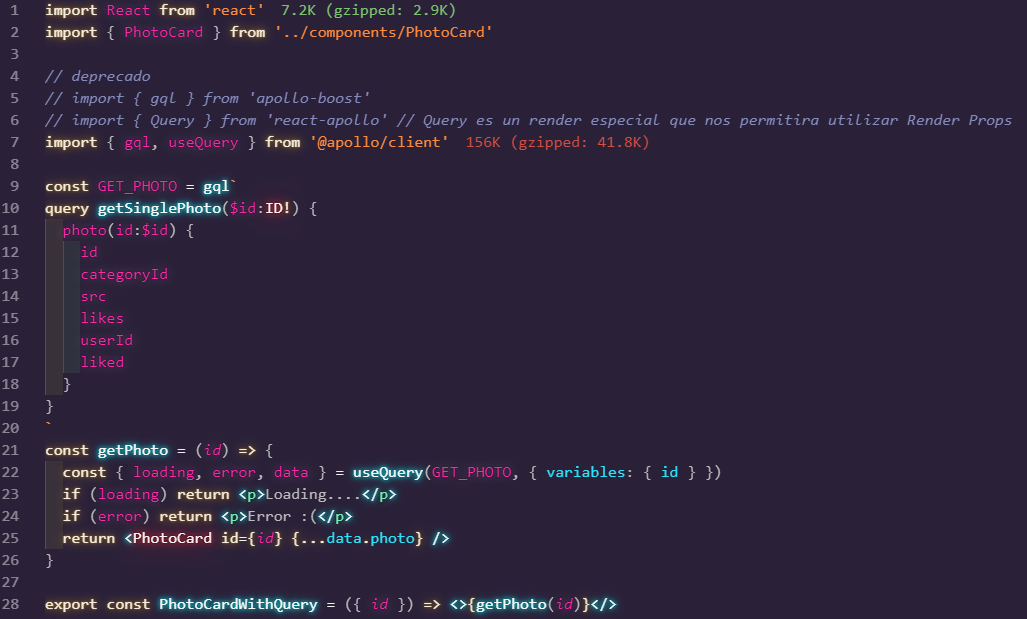
/>

)

}

</GetPokemon>

.  
[Dejo acá el repositorio en el que con 3 componentes básicos (App.jsx, Pokemon.jsx y GetPokemon.jsx) juego con las render props](https://github.com/behagoras/render-props-experiment.git)



Tengan en cuenta tambien que el proyecto react-apollo esta deprecado <https://github.com/apollographql/react-apollo>

warning THIS PROJECT HAS BEEN DEPRECATED warning

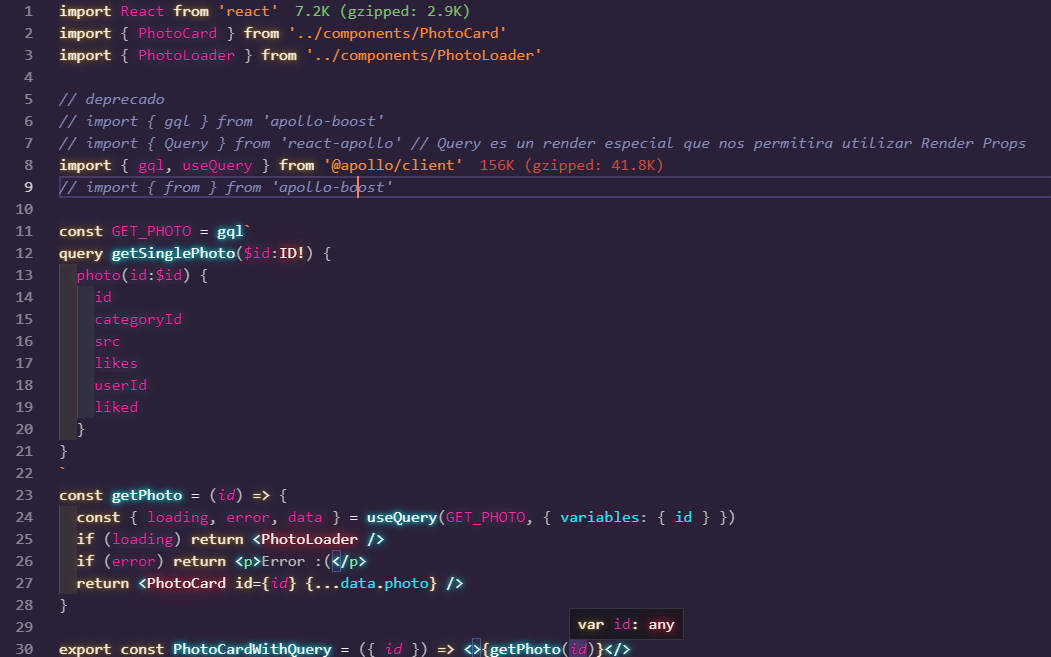
Please note that 4.0.0 is the final version of all React Apollo packages. React Apollo functionality is now directly available from @apollo/client >= 3. While using the @apollo/react-X packages will still work, we recommend using the following imports from @apollo/client directly instead:

ouch! de la manera que estaba siguiendo esta deprecado el component Query 😦

<https://www.apollographql.com/docs/react/api/react/components/>

NOTE: React Apollo’s render prop components are deprecated. They will continue to receive bug fixes until March 2020, after which they will no longer be maintained by Apollo.

# Refactorizando y usando variables de loading y error



Les dejo mi código para los que quieran hacer un efecto de Placeholder loading

deben instalar esta [dependencia](https://github.com/buildo/react-placeholder)

recuerden pasarle la propiedad “loading” al PhotoCard

...

import ReactPlaceholder from "react-placeholder"

import { TextBlock, RectShape, RoundShape } from "react-placeholder/lib/placeholders"

...

const photoCardSkeleton = (

<React.Fragment>

<RectShape color="#eee" style={{ height: "400px", marginBottom: 10 }} />

<div style={{ display: "flex", flexDirection: "row", alignItems: "center" }}>

<RoundShape

color="#eee"

style={{ width: 50, height: 50, marginLeft: 10, marginRight: 10 }}

/>

<TextBlock color="#eee" rows={1} style={{ width: 120 }} />

</div>

</React.Fragment>

)

**return** (

<Article ref={element}>

{show && (

<ReactPlaceholder

ready={!loading}

showLoadingAnimation={**true**}

customPlaceholder={photoCardSkeleton}

>

<React.Fragment>

<a href={`/?detail=${id}`}>

<ImgWrapper>

<Img src={src} />

</ImgWrapper>

</a>

<Button onClick={() => setLiked(!liked)}>

<Icon size="32px" /> {likes} likes!

</Button>

</React.Fragment>

</ReactPlaceholder>

)}

</Article>

)

quedaría así:

# Usando las mutaciones con los likes

El componente de mutation es otro componente importante en una aplicación Apollo. Es un componente React que proporciona una función para ejecutar una mutation de GraphQL para así alterar la *data*. Además, rastrea el estado de carga, finalización y error de esa mutación.

Actualmente la libreria apollo se implementa como @apollo/client y esta super genial usarla como un custom hook. Les comparto el codigo

**import** { useMutation, gql } from '@apollo/client'

const MUTATION\_LIKE\_PHOTO = gql`

mutation likeAnonymousPhoto($input: LikePhoto!) {

likeAnonymousPhoto(input: $input) {

id,

liked,

likes

}

}

`

export const useMuationToogleLike = () => {

const [mutation, { loading: mutationLoading, error: mutationError }] = useMutation(MUTATION\_LIKE\_PHOTO)

return { mutation, mutationLoading, mutationError }

}

Para usarlo lo hacen asi

**import** React from 'react'

**import** { ImgWrapper, Img, Article } from './styles'

**import** { useLocalStorage } from '../hooks/useLocalStorage'

**import** { useNearScreen } from '../hooks/useNearScreen'

**import** { useMuationToogleLike } from '../hooks/useMutationToogleLike'

**import** { FavButton } from '../FavButton'

const DEFAULT\_IMAGE = 'https://res.cloudinary.com/midudev/image/upload/w\_300/q\_80/v1560262103/dogs.png'

export const PhotoCard = ({ id, likes = 0, src = DEFAULT\_IMAGE }) => {

const [show, element] = useNearScreen()

const { mutation, mutationLoading, mutationError } = useMuationToogleLike()

const key = `like-${id}`

const [liked, setLiked] = useLocalStorage(key, false)

const handleFavClick = () => {

!liked && mutation({

variables: {

input: { id }

}

})

setLiked(!liked)

}

// console.log('{ mutation, mutationLoading, mutationError }', { mutation, mutationLoading, mutationError })

return (

<Article ref={element}>

{

show && <>

<a href={`/?detail=${id}`}>

<ImgWrapper>

<Img src={src} />

</ImgWrapper>

</a>

<FavButton

liked={liked} likes={likes}

onClick={handleFavClick} />

</>

}

</Article>

)

}

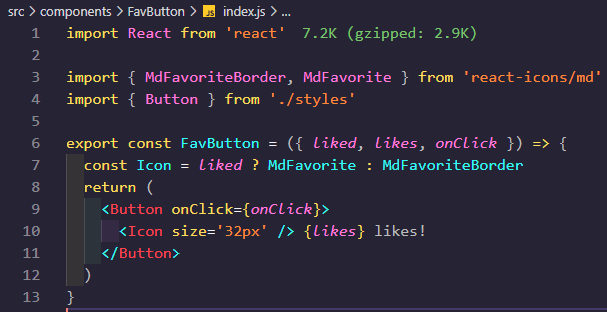
Seguramente te estás preguntando cómo puede ser esto, pero si no hemos cambiado en ningún sitio, que tengamos que incrementar el número de likes.

Si te fijas en el componente, en ningún sitio hemos tocado nada de las props para que el número de los likes se vaya incrementando cada vez que lo hacemos.

Esto es parte de la magia que nos ofrece apollo en la mutación.

Si te fijas, hemos devuelto el Id del detalle que nos está gustando el Id de la foto que nos gusta y no lo estamos usando para nada.

¿Entonces, para que lo hacemos? Pues lo hacemos porque internamente react apollo detecta, que esta Id está sufriendo una mutación y está actualizando la caché y hace que estos cambios se vean renderizados en la UI de forma totalmente gratuita, sin que tengas que hacer nada.



Reach router

# Usando las mutaciones con los likes

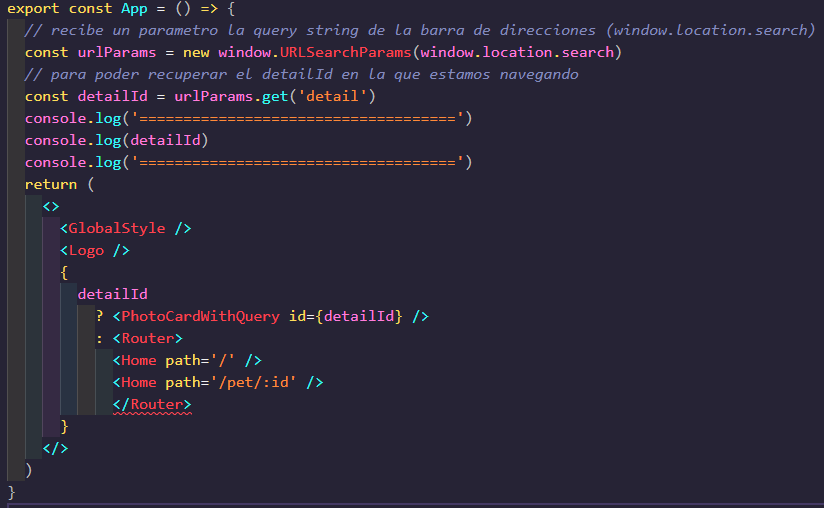
Reach Router es una versión simplificada y mejor optimizada de React Router, su creador es Ryan Florence el mismo creador de React Router. Se anunció que los dos paquetes se iban a unir, pero su API se va a parecer más a Reach Router.

En este módulo te vamos a enseñar como usar este increíble paquete para manejar las rutas en nuestra aplicación.

npm install @reach/router

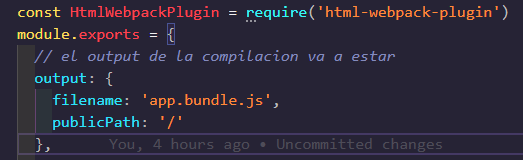
App.js











Para hacerlo en webpack se necesita agregar dos cosas, el publicPath en output e historyApiFallback en devServer:

output: { publicPath: '/', }, devServer: { historyApiFallback: true },

ListOfCategories Index.js



# Usando Link para evitar recargar la página

En esta clase utilizaremos el componente Link para hacer que nuestros enlaces no hagan que la página se recargue y funcione como una Single Page Application. Recuerda siempre que puedas utilizar Link en lugar de a para tener una mejor experiencia de usuario y que no recargue la página.

