**REACT AVANZADO**

React es actualmente la tecnología web más demandada en el mercado laboral y presenta un sinfín de herramientas para crear vistas de aplicaciones complejas, desde react native para aplicaciones móviles nativas, hasta realidad virtual.

**¿Por qué debes seguir profundizando en tus conocimientos de React?**

1. Actualmente es la tecnología más demandada del mercado.
2. Divide el código utilizando React.lazy.
3. Render props.
4. Hooks personalizados.
5. Comparte la lógica con componentes de orden superior.

En este curso vamos a ver conceptos más avanzados de react y construiremos una aplicación desde 0 llamada petgram, es como un Instagram pero de mascotas.



Tendremos nuestras rutas, gestión de usuarios y likes. Utilizaremos como empaquetador y transpilador:

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

Estilado con CSS en JS con:

* [styled-components](https://www.styled-components.com/) que nos permite olvidarnos de las clases para evitar colisión, nos genera hashes únicos a nuestros elementos

Como linter utilizaremos:

* [Standard JS](https://standardjs.com/) que consiste en una serie de reglas pre establecidas para empezar rápidamente nuestro proyecto y asegurar máxima calidad

Para fetching (obtención) de datos:

* [GraphQL](https://platzi.com/cursos/graphql/) y [React Apollo](https://www.apollographql.com/docs/react/) para el manejo de los datos, el backend ya esta creado

Para el enrutado de la SPA utilizaremos:

* [Reach Router](https://reach.tech/router) otra alternativa a react router, es una api muy sencilla de usar.

Para las buenas prácticas utilizaremos:

* [Lighthouse](https://developers.google.com/web/tools/lighthouse/?hl=es) auditoria de la performance y de PWA. [Cypress](https://www.cypress.io/) para testing de la app end to end

Por último haremos SEO, PWA y Deploy con:

* [React Helmet](https://github.com/nfl/react-helmet) para el seo, [Workbox](https://developers.google.com/web/tools/workbox/) para service worker de app offline, [Progressive Web App](https://platzi.com/cursos/pwa-react-js/) y [Deply con Now](https://platzi.com/cursos/deploy-now/)

Para empezar nuestro proyecto instalamos:

1. Webpack & Webpack-cli

npm install webpack webpack-cli --save-dev

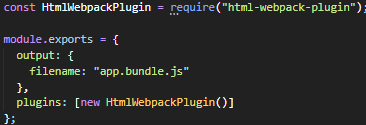
1. Html Webpack Plugin

npm **install** html-webpack-**plugin**--save-dev

1. Webpack dev server

npm install webpack-dev-server --save-dev

Luego de esto creamos nuestro archivo de configuraciones de webpack (webpack.config.js) y agregamos:

En el packaged.json creamos las tareas para producción y desarrollo con dev server

Otra manera de crear un bundle es creando un index.js en una carpeta src y escribiendo en la terminal: ./node\_modules/.bin/webpack src/index.js

Nos creara una carpeta dist con el archivo main.js que tendrá el contenido de index.js

Sin embargo necesitamos crear un HTML y un js que usaremos en la aplicación, así que usaremos los scripts creados en el packaged.json para crear un HTML y un js válidos. Con el comando npm run build creamos el app.bundle.js que tenemos en las configuraciones del webpack como output y además, gracias al plugin de HTML, también se creara un HTML que nos permitirá mostrar la app en el navegador. Con npm run dev el cual esta configurado con el webpack-dev-server nos creara un servidor en el puerto 8080. Ten en cuenta que esto es una configuración súper básica de webpack para empezar el proyecto

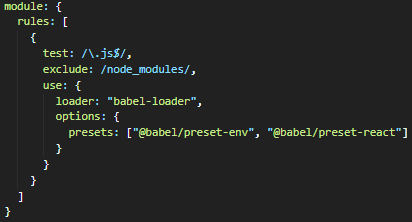
**Instalación de React y Babel**

Ahora instalaremos react, react-dom y babel. Empezamos instalando npm i -D react react-dom y luego creamos nuestro HTML donde estará el script que hará referencia a ese HTML con un div.

Ahora debemos instalar babel, con npm i -D @babel/core @babel/preset-env babel-loader los cuales nos ayuda para que los navegadores entiendan nuestro código jsx de react.

También instalamos el preset de babel react npm i -D @babel/preset-react

Y en nuestra configuración de webpack luego de los plugins:

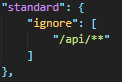


Ya con esta configuración podemos escribir código jsx que lo entenderán todos los navegadores

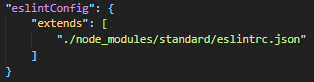
**Linter, extensiones y deploy con Now**

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:



* Ahora, queremos que nuestro Linter nos detecte los errores a medida que vamos escribiendo, para hacer esto añadimos lo siguiente a nuestro package.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: quitamos el check 
  + Prettier: ESLint Integration: checkeamos 
  + Eslint: Auto Fix On Save: checkeamos 

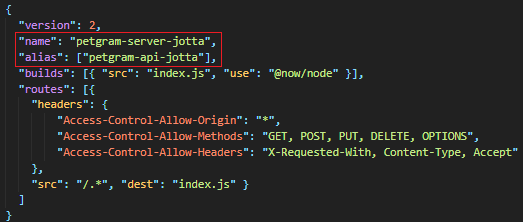
Ahora que tenemos todo el entorno preparados vamos a deployar tanto el servidor como el proyecto, ¿qué significa esto? El caso es que el proyecto es únicamente frontend (el backend ya está creado) pero como tenemos una carpeta “api” donde esta todo nuestro backend, esta debe estar en internet si lo queremos utilizar. Para ello vamos a utilizar now, el cual es una plataforma que nos permite deployar de forma muy sencilla nuestro proyecto.

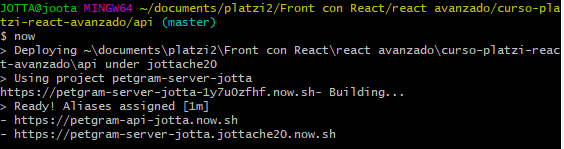
Entramos en la carpeta api, la cual ya posee nuestro backend creado y además un archivo llamado now.json este archivo es el que desplegara nuestro backend a internet.

Primero debemos instalar now de manera global en nuestro sistema para eso usamos npm install –g now luego de esto debemos acceder a la página web de now e iniciar sesión con nuestra cuenta de github. Una vez lo hacemos y seguimos los pasos debemos iniciar sesión en nuestro ordenador, para esto en la terminal usamos el comando now login [email@gmail.com](mailto:email@gmail.com) y nos iniciara la sesión.

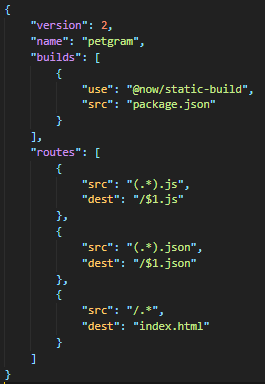
Una vez instalado now, creado la cuenta en la página web oficial e iniciado sesión en nuestro ordenador, vamos a entrar en la carpeta de api y usar el comando now que está disponible ya que instalamos el CLI de now.

Este comando nos creara una URL donde estará nuestro backend, debemos tomar dichas URL y tenerlas guardadas para usarlas más adelante. Recuerda colocarle un name y un alias personalizado ya que solo puede haber una URL con el mismo nombre y ya el profesor del curso la tiene creada.





Ahora lo que necesitamos hacer es deployar la aplicación como tal y pueda verse con una URL. Creamos un archivo now.json en la raíz del proyecto y a esta le hacemos las siguientes configuraciones:

Le indicamos la versión de now que queremos utilizar en este caso usaremos la 2

Le decimos el nombre de nuestro proyecto

La construcción que utilizara, hay diferentes disponibles pero usaremos la de static-build

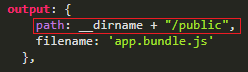
Y esta la tiene que ir a buscar en el package.json

Ahora le pasamos las rutas que tendrá, será una SPA así que siempre tendrá que buscar el index.html asi que el src cuando sea cualquiera ira al index.html

Además también queremos servir los archivos.js y le decimos que cualquiera que sea su ruta con la extensión .js le decimos que el destino será el mismo pero con js

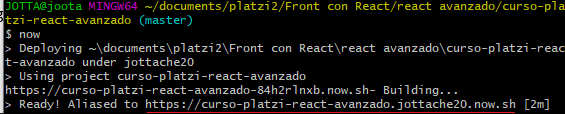
Lo mismo hacemos con los Json

Luego en el webpack.config.js agregamos



Luego debemos agregar un nuevo script en el package.json este será: 

Ahora podemos usar el comando now en la raíz de nuestro proyecto, esto hará que nuestra app esté disponible en una URL en internet:



CSS in JS (css en JavaScript)

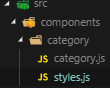
CSS-in-JS nos permite no solo hacer el jsx si no colocar los estilos en el mismo JavaScript. Con css-in-js no debemos preocuparnos por tener dos clases iguales, no nos debemos preocupar por el Critical Path CSS, tendremos un css mucho más óptimo. Esta biblioteca es pensada más que todo para React. Styled-component es la librería que usaremos y es totalmente compatible con material-ui, aquí podemos revisar cómo hacerlas trabajar juntas: <https://material-ui.com/guides/interoperability/#styled-components>

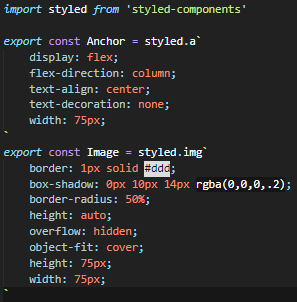
Vamos a construir nuestro primer componente el cual será y usaremos [styled-components](https://www.styled-components.com/) para hacer los estilos de nuestro proyecto.

Lo primero que haremos será instalarlo en entorno de producción. Esto quiere decir que no usaremos el -D. el comando para la instalación es npm install styled-components

Una vez instalado debemos instalar también una extensión de VSCode que nos permitirá visualizar la sintaxis de mejor manera, ya que cuando usemos styled-components tendremos que encerrar los estilos en comillas compuestas, con esta extensión evitaremos que se vea todo verde.

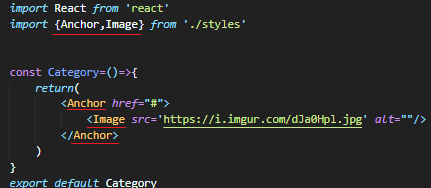
Ahora que lo instalamos creamos un archivo llamado styles.js el cual tendrá nuestros estilos, este archivo debe ser creado en la misma carpeta donde está el componente al que le aplicaremos los estilos.



Una vez creado la sintaxis es la siguiente:

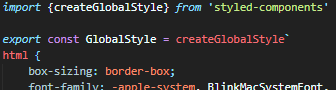
Primero debemos importar la librería de styled-components

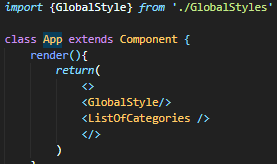
Luego, vamos a crear una constante exportada que tendrá un nombre creado por nosotros, este por convención debe ser camelCase y se le asigna el styled más la etiqueta a la que le aplicaremos el estilo, en este caso se la aplicamos a un link (a) y a una imagen (img) quedaría styled.a y styled.img seguido de comillas compuestas y el css que le aplicaremos a dicha etiqueta.



Luego importamos cada uno de los const con sus estilos que usaremos en el componente y reemplazamos la etiqueta por el nombre que le dimos en el archivo styles.js de manera que este haga referencia a los estilos y a la etiqueta, note que donde debería haber una a y un img se encuentran las constantes que importamos.

Para aplicar estilos globales styled-components nos permite utilizar createGlobalStyle el cual lo importamos y lo usamos de la siguiente manera:

Usamos createGlobalStyle en una constante seguida de comillas compuestas y este solo hay que importarlo en el componente de app, donde se renderiza toda la aplicación (no en el index)



Se coloca encima de todos los componentes a renderizar, es decir de primero, recuerda usar un fragment, un div o un <> </> para encerrar todo

React-Icon

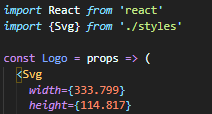
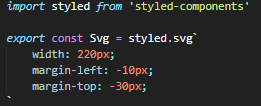
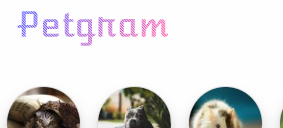
React-icon es una librería de react que nos permite usar una gama de iconos totalmente gratis, solo debemos instalarlo como dependencia de producción con npm install react-icon y en la documentación nos mostrara como usar los iconos. En este caso usaremos los iconos que nos trae de material-ui, importamos el icono y lo llamamos de react-icons/md (todo lo necesario sale en la documentación)



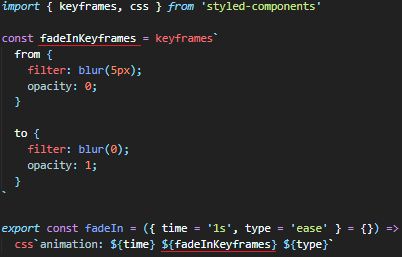
**Crear svg en componente de react**

* 1. Primero entramos en la página maketext.io la cual nos proporciona una cantidad de textos personalizados
  2. Luego entramos en SVGOMG para eliminar texto de relleno
  3. Y para finalizar entramos en SVGR que nos ayudara a convertir el svg en componente

El resultado del paso numero 3 será un componente de react el cual vamos a crear y a este le podemos aplicar estilos para definir su tamaño entre otras propiedades.

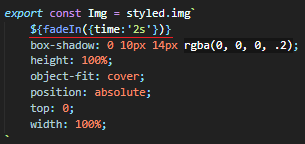
**Animaciones con keyframes en styled-components**

Las animaciones en styled-component se deben importar de la librería de styled-components, importamos los keyframes y una propiedad llamada css que coloca código css en donde lo pongamos

Lo correcto es crear una carpeta llamada styles en donde colocaremos un archivo llamado animation.css que es en donde iran todas nuestras animaciones

Creamos una const y la hacemos igual a keyframes seguido de comillas compuestas que llevara dentro los keyframes tal cual como se haría con css vanilla.

Luego creamos una const que contiene una función, esta función lleva por parámetros las propiedades de animation las cuales serán dinámicas, como por ejemplo time e ease. Esta función será la que exportaremos y recibimos en el archivo de styles donde queramos colocar dicha animación se llamara con el nombre de la función.

Esta función puede recibir parámetros para cambiar el tiempo, type, entre otras propiedades de la animación.

Estos parámetros deben lleva la siguiente sintaxis: función({ time: ‘2s’})

Esto lo hacemos si queremos hacer una función que haga una animación y pueda ser reutilizable, sin embargo para animaciones por individual en componentes, solo usamos los keyframes y colocamos las propiedades de animation en donde queramos aplicarla con el nombre de la animación en ${}

Podrías copiar el css de animate.css y crear funciones reusables

**HOOKS**

React hooks es una nueva característica de react que nos permite hacer un monton de cosas, son una característica nueva asi que solo está disponible desde la versión 16.8.0

Los hooks (ganchos en inglés) como su propia palabra lo dice, te enganchan a funcionalidades de react que hasta hace poco no podíamos utilizar en componente funcionales, en otras palabras son funciones que nos permiten acceder a casi todas las características de React desde componentes funcionales

¿Por qué casi todas? Porque todavía no podemos usar el 100% de las funcionalidades, como por ejemplo aun no podemos utilizar el ComponentDidMount o el getSnapshotBeforeUpdate por ahora.

Los hooks mas importantes y los que utilizaras en el 90% de los casos son:

* useState para añadir un estado local en tu componente
* useEffect que nos permitirá ejecutar una función cada vez que rendericemos nuestro componente
* useContext que nos permite acceder a la context API para tener valores que utilizaremos en toda nuestra app de forma global sin necesidad de pasarla por las props

Además tenemos otros hooks auxiliares que nos permiten acceder a otras funcionalidades, tales como son:

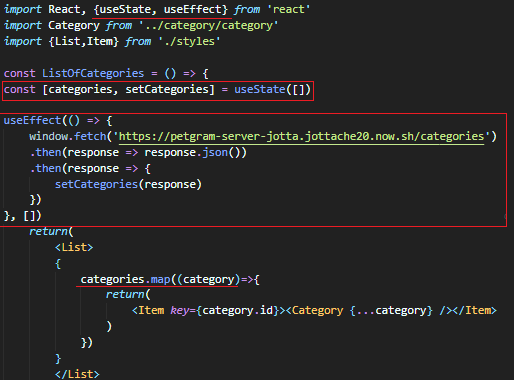
* useReducer
* useCallback
* useMemo
* useRef
* useImperativeHandle
* useLayoutEffect
* useDebugValue

Custom hooks puedes crear tu propio hooks para por ejemplo, evitar escribir código repetido en diferentes componentes

Ventajas:

* 100% recompatible, no tienes que modificar demasiado tu código react para implementar los hooks.
* Mejor compilación con babel
* Mejor performance, ya que es mucho más rápido ejecutar una función que tener que evaluar una clase

Ahora usaremos hooks para manejar un estado y un llamado a la API que creamos y deployamos con now.

Lo que hacemos aquí es usar useState para crear un estado, useState es igualado a un array de 2 elementos, que hacen referencia al estado (primer elemento del array) y al estado seteado (segundo elemento del array) luego igualamos este array a useState() que recibirá por parámetro un array vacío, esto es importante porque cuando estamos haciendo un map tiene que ser de un array, y si no colocamos el array vacío no podría hacer el map y la app se rompería antes de hacer la promesa.

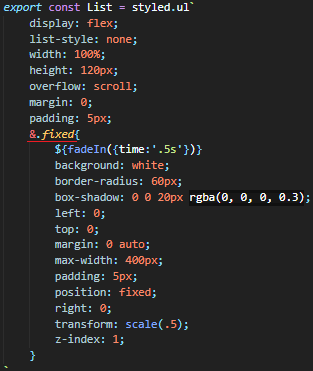
Luego usamos el useEffect que ejecuta una función luego de que el componente se monte, este se montara sin elementos gracias al array vacío, sin embargo apenas se monte llamara a esta función, la cual realiza un fetch, este debemos llamarlo de window ya que estamos usando la sintaxis de .then Luego convertimos la respuesta en un Json y se la metemos al setCategories el cual es el hook que setea el estado. Debemos ponerle de segundo parámetro al useEffect un array vacío que evitara que se realice esta función infinitamente

**Creando un elemento que aparece según el estado y el scroll del usuario**

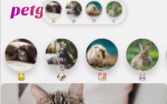
Queremos crear una lista de categorías que aparece cuando el usuario hace una determinada cantidad de scroll.

Lo primero que hacemos será crear una función que retorne la lista, y le pasamos un parámetro llamado fixed (puede ser otro nombre) y a la lista le pasamos un className que dentro tendrá un operador ternario donde evaluara si ese parámetro llega (true), entonces agregara un string ‘fixed’ (es decir le agregara una clase) y sino (false) le agregara un string vacío

Luego en el return que es donde mostramos el componente vamos a ambos casos, una con el parámetro true para que se muestre y tenga la clase fixed y otra sin parámetros para que se muestre normalmente.

En los estilos vamos a agregar un &.fixed el cual hace referencia a los estilos que posean ambas clases, en este caso aplicara al componente que llegara con la clase fixed.

Se mostraran ambas listas de categorías, una como normalmente y otra con la clase fixed agregada:

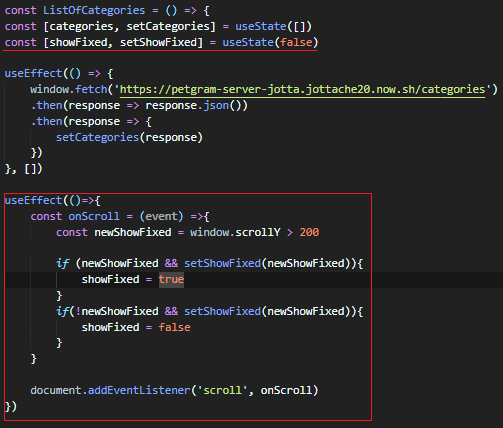


Como buena práctica para no usar className, styled-components puede recibir propiedades, así que podemos mandársela y los estilos evaluamos si llega



Dentro de los estilos del Link colocamos un template string para evaluar las propiedades que llegan, y usamos la propiedad css para colocar el css dentro de comillas compuestas

Ahora queremos hacer que se muestre dependiendo del scroll del usuario.

Para esto creamos otro useState y otro useEffect.

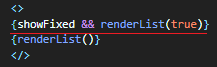
En el useState vamos a pasarle el estado en false y en el useEffect es en donde tendremos el control del scroll.

Creamos una función dentro del useEffect y dentro de esta vamos a meter en una constante un window.scroll mayor a 200, esto quiere decir que si el scroll del usuario supera los 200px será true, de lo contrario será false.

Luego evaluamos con un if si la constante que hicimos anteriormente y el ‘setShowFixed’ (seteador de useState) llegan en true, es decir, si el scroll del usuario es mayor a 200, entonces el estado ‘showFixed’ del useState será true

Y en otro if evaluamos si ambos llegan en false entonces el estado del useState se mantendrá el false como ya viene predefinido por nosotros.

Luego agregamos el document.addEventListener(‘scroll’, onScroll) el cual llamara a la función que creamos cada vez que el usuario haga un scroll.

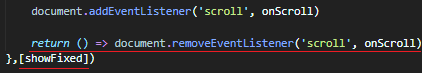
Ahora debemos mostrar la lista únicamente si showFixed (estado del useState) y el renderList() llegan con true De manera que si llega el false el estado no se mostrara.

Así se verá al superar los 200px de scroll:



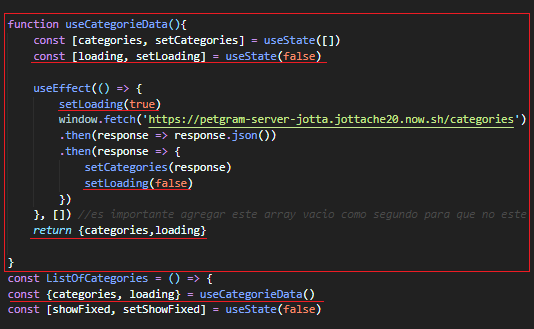
nota: esta es una forma mucho más resumida de realizar los dos if que hicimos anteriormente. (No la entiendo)

Podemos agregar para más performance lo siguiente:

Quitamos el escuchador de eventos cuando no haya más scroll del usuario Y para que se ejecute el efecto cada vez que cambie el estado y no constantemente

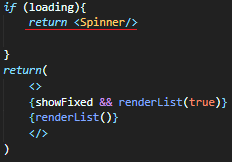
Custom hooks

Podemos crear custom hooks que realicen todo lo que deseemos hacer

Aquí tomamos el mismo hook que nos hacia el fetching de los datos y lo metimos dentro de una función que esta por fuera de la función principal del componente, esta función posee el useState y el useEffect para traer los datos del fetch y además agregamos un useState para manejar el estado de loading, este por defecto estará en false.

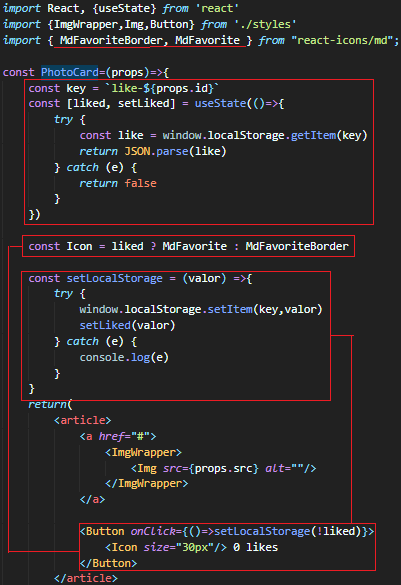
Dentro del useEffect seteamos el estado del loading a true, el cual se mantendrá así mientras se hace la promesa, este mismo estado vamos a volver a cambiarlo a false cuando llegue la promesa, en la línea donde seteamos las categorías también seteamos el estado de loading.

Recuerda que un hook devuelve algo, ósea debe llevar un return de sus datos o sus estados, estos los metemos dentro de llaves para destructuring

Luego los llamamos dentro de la función principal del componente, llamando la función que creamos arriba que sería nuestro custom hook y asignándoselos a una constante entre llaves. Ahora estarán disponibles estos datos dentro del componente

Ahora podemos usar el estado de loading para cargar un spinner si deseamos darle una experiencia al usuario más agradable

Manejando el like de las fotos (cada foto tendrá su like por separado)

Lo primero que hacemos es crear un const donde almacenaremos un string con el id de cada foto que recibimos por las props.

Una vez lo tengamos creamos un estado con useState, este estado almacenar el like de la foto, este useState recibe una función la cual realiza un try catch, dentro del try creamos una const que acceda al locaStorage con el método getItem para obtener un elemento del storage y que este elemento lo retorne con un JSON.parse para que lo transforme en un número, recuerda que el storage almacena un Json con keys y valores como string. En el catch retornamos false para que se mantenga en dislike la foto, usamos un try Catch porque si la navegación esta privada no se puede acceder al locaStorage.

Ahora debemos darle ese valor al localStorage para que el useState pueda recuperarlo, para esto creamos otra función que también tiene un try catch ya que accederemos al localStorage, usamos el window.localStorage para acceder y le pasamos el método setItem con su key y valor, le pasamos este valor al estado. El estado empieza en false la primera vez por el catch del estado, y esta función que acabamos de crear setea el valor de ese estado que llega por parámetros, ahora debemos enviarle este parámetro.

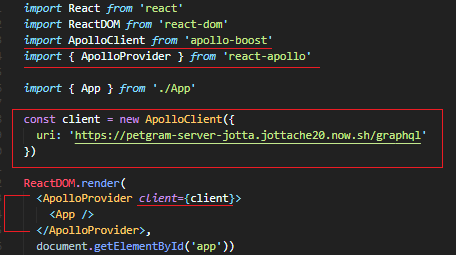
Para enviárselo en el botón llamamos a la función que acabamos de crear y le pasamos como parámetro el estado cambiado, es decir si está en false, pues pasara true a la función, y viceversa. Para finalizar hacemos un operador ternario que cambie el icono dependiendo del booleano que le llegue del estado. Si es true será relleno, si es false será vacío.

**GraphQL y React Apollo**

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.

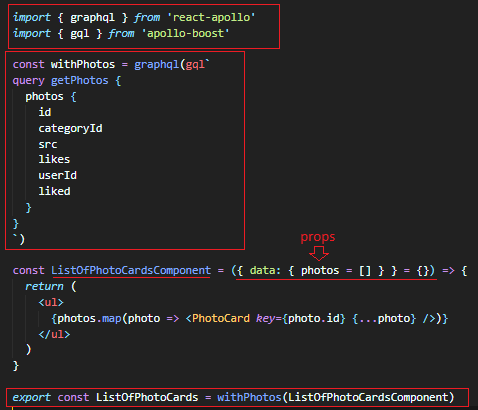
Vamos a instalar npm i apollo-boost, una utilidad que nos permite inicializar nuestra conexión con un servidor de Apollo muy rápidamente y sin configuración, necesitaremos npm i react-apollo el cual es la integración de Apollo con React y finalmente npm i graphql.

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

Lo que estamos haciendo aquí es crear el cliente donde estará nuestra configuración de apollo, esta que nos proveerá de la API REST donde está el Json con todos nuestros datos.

Y el provider que nos va a proveer nuestra aplicación de los datos de esa API, funciona igual que el provider de Redux, solo que en los componentes donde usaremos la API vamos a hacer algunas modificaciones de graph.

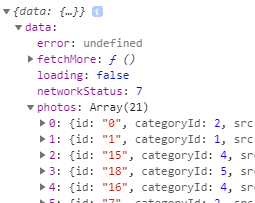
Si quisiéramos integrarle Redux a la app, solo tenemos que agregarle el provider de Redux y encerrar la aplicación entre ambos providers (el de apollo primero)

En el componente donde usaremos los datos importamos a graphql para usar su API de métodos y gql que nos permitirá escribir la query que usemos en la página.

En una const usamos el método y escribimos la query con los datos que usaremos. Graphql nos permite seleccionar solo los datos que queramos usar, en este caso solo usaremos los datos de la key “photos" que están en la API REST.

Luego cambiamos el nombre de nuestro componente, para no confundirnos en el export que haremos luego. Este componente le llegaran los datos por props, sin embargo hay que tomar en cuenta que estamos esperando datos de una promesa, así que lo que debemos hacer es verificar los datos que llegan con un console log





Una vez llegue la promesa notaremos que contiene un objeto llamado data y dentro de este un array llamado photos, estos vamos a pasarlos por props al componente. Para esto vamos a destructurarlo, hacemos al array photos = [] para que se renderize aun si no llega nada y no falle la app. Hacemos lo mismo con data = {} de esta forma la app no falla mientras espera la promesa.

Para proveer al componente de los datos de graphql le pasamos el componente a la const donde están almacenados todos los datos. Todo esto lo metemos dentro de export const



Para filtrar por categoría:

Ya en el servidor está preparado que podamos pasarle a la query una propiedad para que filtre por categoría, esta es la siguiente:



WithPhotos, el cual es el componente donde se realiza la query, es en realidad un componente de orden superior, es decir que envuelve el componente presentacional, el componente resultante de todo esto que sería “listOfPhotoCards” puede recibir por props los parámetros por los que vamos a filtrar

 De esta manera ya puede recibir la categoría por propiedades del padre. Solo mostrara las fotos que contengan el categoryId 3.

Render props para recuperar datos

Ya hemos visto el patrón de utilizar HOC (high order components) para recuperar los datos y envolver un componente para inyectar estos datos y pasárselos por props. Pero existe otro patrón más común que se llama render props

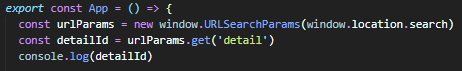
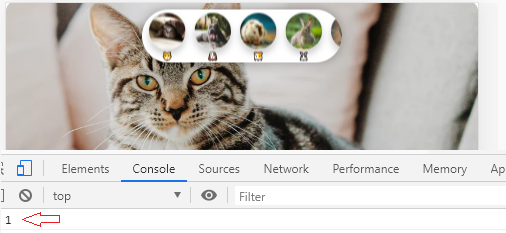
Lo que hacen es convertir la props especial children y en lugar de renderizar un elemento renderiza una función, esta función lo que debe devolver es el componente que queremos renderizar, la gracia está en los parámetros que recibe esta función.

La render props, en la función children, lo que va a recibir por parámetros es la info que queremos inyectar en ese componente que queremos renderizar.

Primero:

Cuando hagamos click en una photocard tiene que acceder a una URL específica, nosotros mismos se la creamos con su href

 Donde le decimos que vaya a un path llamado detail con el id que le llegara por parámetros al componente.

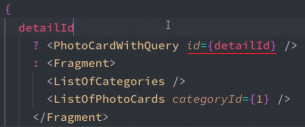
URLSearchParams recibe un parámetro que en este caso sería la url que tengamos en ese momento, así podremos acceder a ella, y ahora lo que haremos será usar el método get para obtener el id al que estamos llamando por la URL. Hacemos un console log para visualizarlo.



Haciendo click en el segundo photocard nos devolverá por consola el id de esa photocard según la URL.

Creamos un componente el cual va a recibir la query, importamos gql y el componente query especial que nos permite utilizar la técnica de render props

Este componente query lo que hace es renderizar una función, por ende debemos pasársela y esta función lo que hará es devolver el componente que queremos mostrar. Este componente query también recibe un parámetro llamado query, el cual será al constante que almacena la query que realizamos con GraphQL.

En app.js vamos a renderizar este componente dentro de un operador ternario, si llega detailId entonces renderizara este componente, sino renderizara listOfPhotoCard y le pasamos ese id por propiedad

Luego debemos volver al componente nuevo que creamos y le pasamos el id a la función pHotoCardWithQuery y al componente query se lo pasamos en una propiedad llamada variables

Sin embargo al darle click a un photocard no estamos recibiendo la foto que le pedimos, sin embargo si está llegando la data, así que ahora debemos pasárselo al componente. Para esto vamos a pasarle a la función que esta renderizando el componente Query esos datos, además le pasamos todo lo que esos datos tiene, lo hacemos con …

Ahora si estaremos recibiendo la foto que queremos respectivamente del id que le pasemos por props.

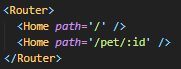
**REACH ROUTER**

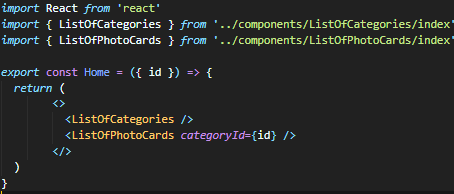
Reach Router es una versión simplificada y mejor optimizada de React Router, su creador es Ryan Florence el mismo creador de React Router. Se anunció que los dos paquetes se iban a unir, pero su API se va a parecer más a Reach Router.

Para instalarlo usamos el comando npm install @reach/router

Y para usarlo

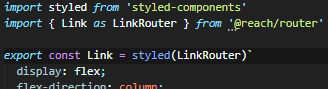


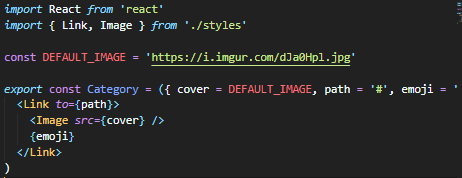
En este caso usamos otra ruta a home con un path diferente, ya que este vendrá con un id que viene por propiedades para que nos mande a la ruta donde aparecen los datos (Cards) de una categoría en específico, la cual es una props que se la pasamos a la lista de categorías. :id es un segmento especial que queremos capturar de una ruta.

Ese id que le pasamos a la ruta será el id que le llegue como props al componente Home, el segmento que le pasemos al componente debe ser el mismo que el componente reciba como props dentro de llaves, es decir si pasamos :id debe llegar {id} si le pasamos :perro debe llegar {perro}



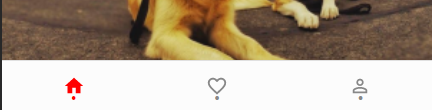
Ahora usaremos el componente link para convertir nuestra app es una SPA.

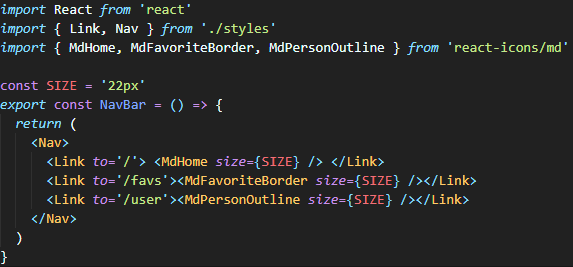
Lo primero que hacemos será configurar el styled-component para que mantenga los estilos que le hemos definido. Para esto en el archivo donde tenemos los estilos, importamos Link y le damos otro nombre (como buena práctica, puedes no darle otro nombre), el método styled puede recibir parámetros, a este le pasamos el nombre que le dimos a Link (o colocamos el link en caso de que no le hayamos colocado otro nombre)

Luego importamos el Link desde el archivo de los estilos, y lo usamos como ya veníamos usando los archivos del styled-component

No olvides que el componente link recibe un to en lugar de un href, el cual indita la ruta que tomara. (En este ejemplo estamos pasando una ruta desde las props, no puede llegar vacía así que le colocamos un #)

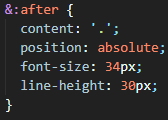
Creando la barra de navegación que aparecerá en la parte inferior de la app:

Para esto creamos un componente que tendrá los diferentes links a cada una de las rutas de la barra de navegación, estos serán estilados con styled component como venimos haciendo en toda la app

A este componente de navegación le vamos a agregar los iconos que nos provee react-icon, ademas le daremos un tamaño por defecto a cada uno desde la propiedad size de manera que no sea necesario hacerlo en el css y podamos modificarlos a todos por igual por medio de una constante que contiene el tamaño en pixeles.

También le agregamos las rutas a donde se van a dirigir

Reach Router nos agrega un atributo aria-current="page" en la página que esté activa en ese momento para que podamos estilarla, agregarle algún tipo de funcionamiento o decirle al usuario en donde se encuentra.

También podemos agregarle un punto que nos indica en cual ruta estamos ubicados.

Podemos agregarle animaciones y demás a todo esto.

**Rutas protegidas**

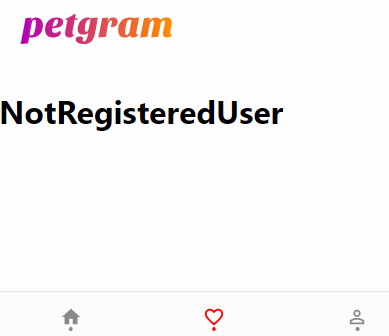
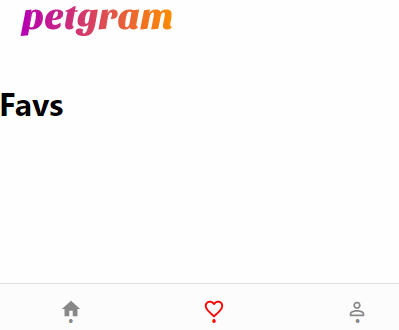
Con el método *render props* realizaremos rutas de autenticación para saber si el usuario ha iniciado sesión o no y podremos mostrar contenido dependiendo de las *props* que le estemos pasando.

creamos una render props donde le pasamos a la función children un json con un key valor que por defecto será en false.

Esta render prop luego la renderizmos en la app con una serie de condiciones: siisAuth (que es el key) llega en true entonces va a renderizar un Router con dos componentes y mostrara al usuario información de esos componentes y solo esa info la cual hace referencia a un usuario registrado.

Si llega en falso mostrara al usuario el componente de NotRegisteredUser con las mismas rutas que las anteriores, estas mostraran información al usuario pero no registrado.

No registrado: registrado:

React.Context

Context API es una característica que tiene React para poder pasar datos en nuestra aplicación sin necesidad de usar las Props.

Para crear un contexto vamos a importar el método createContext de la librería React. El contexto que creemos no va a dejar de ser un componente de React, por ello debe llevar mayúscula al inicio.

El Context que creemos nos va a proporcionar 2 componentes:

* Provider: componente que debe envolver a nuestra aplicación.
* Consumer: nos va a permitir acceder a las *render props* que declaremos en el Provider.

Vamos a simular un inicio de sesión para mostrar las rutas y componentes adecuados dependiendo si el usuario inicia sesión o no.

Creamos un archivo js con el nombre de Context.js, a este archivo vamos a importarle React (ya que usaremos jsx) createContext (para crear el contexto) y useState (para manejar el estado de autenticado o no autenticado)

Primero creamos una constante donde metemos el método createContext()

Luego vamos a crear una render prop la cual poseerá el estado que en principio estará en false.

Luego un objeto que será el value que le pasemos al Provider, este objeto tendrá el estado de useState y una función que lo pasara a true cuando se llame

Luego retornamos el jsx con el Context.Provider y le pasamos la propiedad value con el objeto que creamos. Recuerda colocar los children ya que este Provider encerrara la app.

Para finalizar esta parte, vamos a exportar un objeto con el Provider que seria la render prop y un consumer con el context.consumer

Luego en App.js importamos el contexto



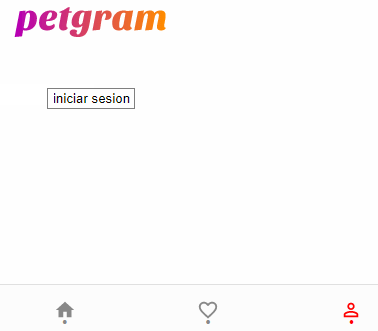
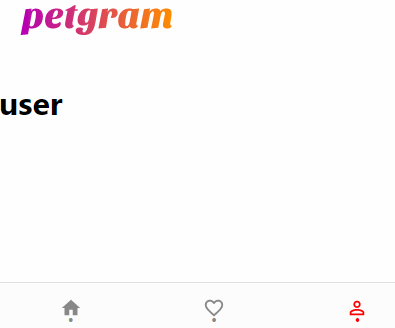
Y lo único que tenemos que hacer es encerrar la parte de la app que renderiza las páginas de favoritos y User con el context.consumer, ya que este es el que va a consumir el contexto que acabamos de crear, este por igual recibe el estado que usamos en el contexto, este seria el isAuth para verificar si llega true o false.

creamos un form dentro del componente que se renderiza en caso de que llegue el estado en false (isAuth) este componente debe tener importado el contexto.

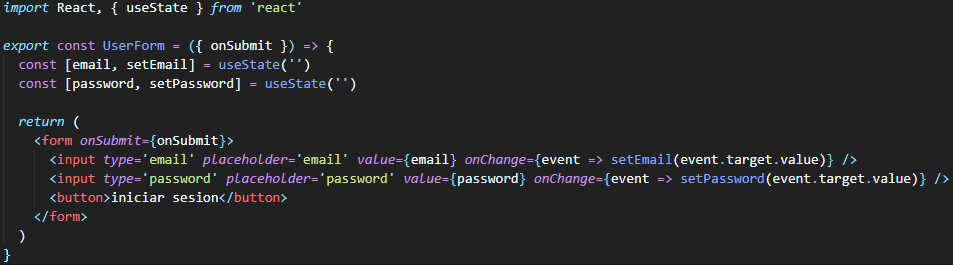
Luego le pasamos el context.consumer el cual le proveerá del value que creamos en el contexto. De este objeto value vamos a pasarle la key que tiene la función que cambiaba el estado a true y retornamos un botón que al hacerle click llama a esa función y cambia el estado a true.

Una vez esto sea true en la app donde se evalua si llega isAuth como true o como false cambiara el componente que se muestra dependiendo de ese estado:

Si llega false: si llega true:

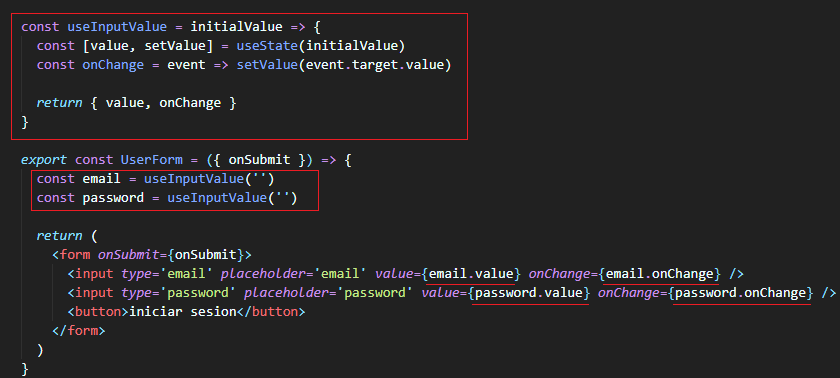
Lo correcto seria crear un componente donde este dicho formulario y pasarlo al componente que se muestra cuando el usuario no este autenticado o no haya iniciado sesión:



Creamos el formulario al cual le pasamos por props el onSubmit, este es el que lleva la función que cambiara el estado a true.

Usamos useState para obtener y almacenar los datos del email y la contraseña, inicializamos estos estados con un string vacio obviamente ya que se trata de strings.

En cada input colocamos los valores referidos como se usan normalmente en los formularios, el form debe llevar su método onSubmit para cuando se envie y los inputs deben llevar el value con el estado actual y un onChange que va leyendo lo que el usuario escribe y lo almacena en el seter de cada estado del useState para actualizar el estado

Para hacer el mismo código pero de una manera más reutilizable vamos a crear un custom hook el cual va a almacenar en un estado el valor y el método que almacena lo que escribe el usuario. Esto lo hacemos porque en ambos input se repite el value y el onChange.

Este custom hook utiliza un useState para almacenar y cambiar el estado y una función onChange que almacena lo que escribe el usuario.

Luego debemos exportar este estado y el onChange para poder usarlo en el jsx.

En el componente ahora vamos a crear dos constantes donde llamamos el custom hook y le pasamos como parámetro un string vacío.

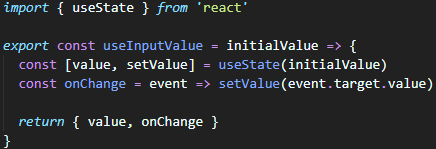
Ahora el value y el onChange de los input solo debemos pasarle el nombre de la const + el método del custom hook, ej.: email.value y email.onChange

Nota que eso hace exactamente lo que hacía el código anterior pero más simplificado.

Luego solo debemos usar este componente en el componente que se renderiza cuando el User no está autenticado o sin sesión iniciada:

Le pasamos al componente UserForm la propiedad onSubmit con la función que cambia el estado de autenticación, este componente lo recibirá para poder hacer el onSubmit adecuadamente.

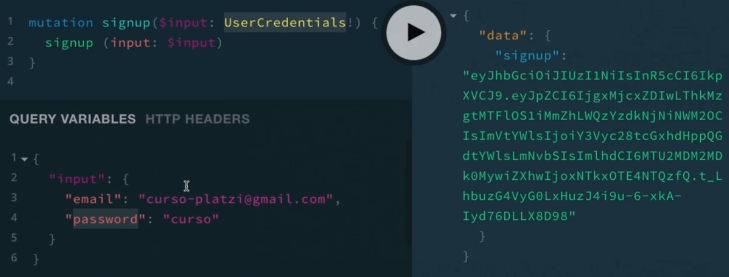
Lo correcto tambien es meter todoso los custom hooks en la carpeta de hooks, asi que nos lo llevamos para alla y para usarlo solo debemos exportar el hook e importarlo en el componente:



**Mutaciones para registro**

De momento nuestro formulario no está guardando la información de registro en ningún lado, para ello vamos a usar un Mutation de GraphQL para mandarle la información a nuestro backend y que nos regrese un JSON Web Token.

Para configurar esta Mutation vamos a crear un container llamado RegisterMutation.js con el siguiente código:

este graphQL lo que hace es devolver un web token a partir del mutation el cual es un metodo de grapQL que lo que haces es enviar datos al servidor y de la funcion signup la cual ya esta definida en el backend y es la que devuelve un json web token. En donde la funcion signup va a recibir por parametros una propiedad input y los UserCredentials. Los UserCredentials son los datos que Le pasamos por http (por el form de la app en el navegador) en este caso el objeto input con el email y el password y al ejecutar esto devuelve un json web token

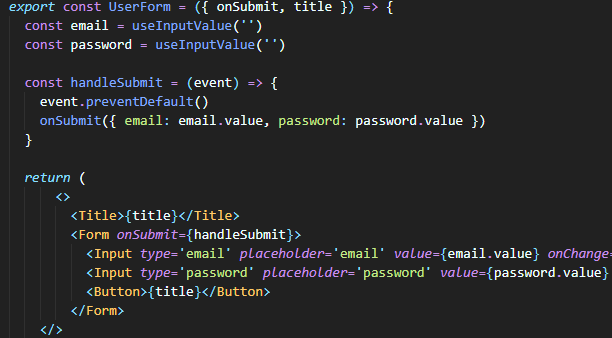
Luego importamos react, Mutation y gql en el componente que creamos

Vamos a escribir el código que nos devuelve el token y lo metemos en una constante REGISTER

Luego vamos a crear el render prop el cual retornara un componente Mutation que sacamos de react-apollo este recibe una propiedad Mutation que envía al servidor lo que le pasaremos, en este caso el REGISTER, este componente va a renderizar a todos sus hijos pues va a encerrar el formulario que creamos de registro

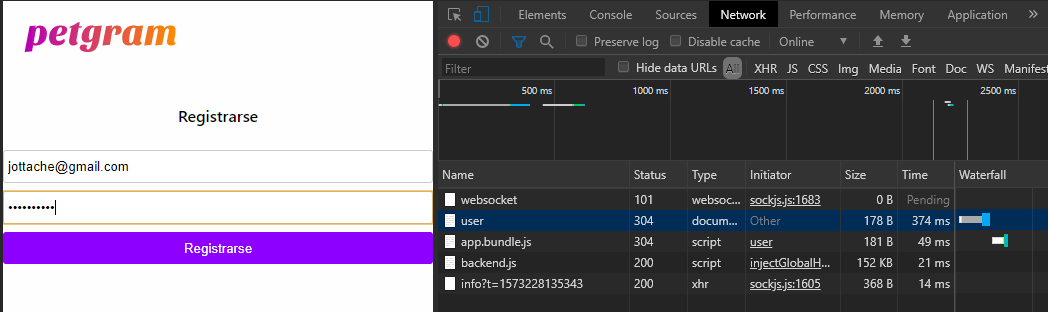
el componente Mutation recibe una función asi que debemos abrir llaves para usar js, esta función nos devuelve el parámetro para poder hacer la mutacion el cual va a ser register y retorna el formulario ya creado.

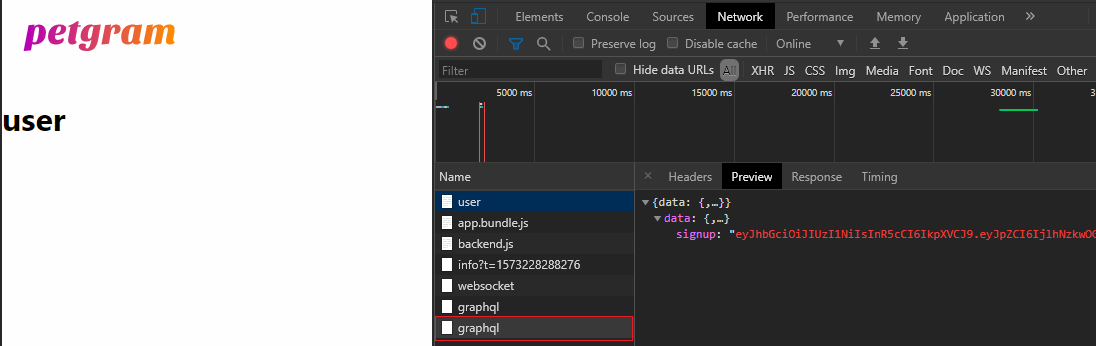
Luego crearemos un onSubmit personalizado, este vendrá desde el UserForm y recibirá como parámetros las props del email y el password que le enviamos desde el formulario de registrarse, estas props las metemos en una const y luego esa constante la metemos dentro de una constante llamada variables ya que el método “register” que es un método heredado de Mutation, debe recibir una propiedad llamada “variables” luego hacemos un .then (ya que este método register es una mutacion que devuelve una promesa) y le pasamos activeAuth el cual será true para autenticar el usuario

en UserForm vamos a crear un handleSubmit de manera que vamos a controlar el evento que le pasaremos a la función useForm.

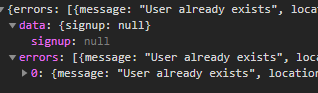
Lo primero que hacemos es crear un preventDefaul() para evitar recargas de la pagina, luego al onSubmit le vamos a pasar los datos del value de los input ya que son los datos que necesitamos, a estos lo renombramos y lo pasamos como un objeto.

Luego se lo pasamos al form por la propiedad onSubmit

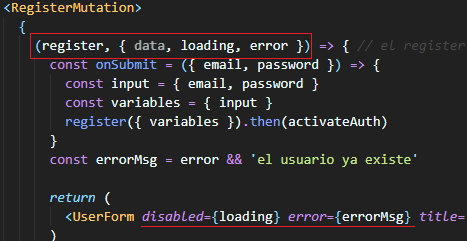
En la app registramos un usuario en el formulario

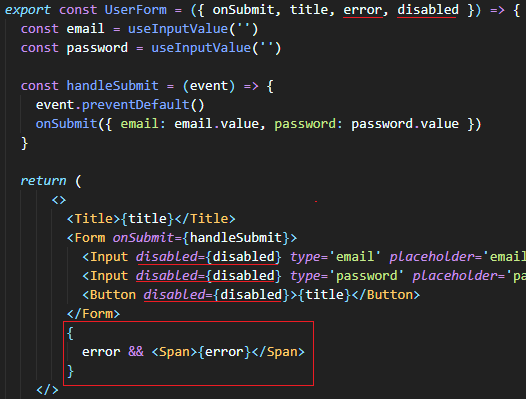


Al ser efectivo el registo este nos devuelve el json web token

Si le pasamos el mismo usuario en email (o alguno de los dos campos) nos dira que ya esxite. En estos casos se puede controlar los estados de error y cargando.

Para ellos vamos a pasarle dos propiedades más que nos llega a nuestra render props de RegisterMutation que llega por la función, en esta además de register (mutacion) tambien nos llega en caso de que el usuario ya este registrado, los datos de error y loading

Vamos capturarlos en los parámetros que llegan y los vamos a pasar como propiedades en el UserForm, el estado de carga lo pasaremos como un disabled el cual es una propiedad de los formularios para deshabilitar el uso de los inputs y botones. Y el error lo manejamos con una constante que evalúa si llega el error entonces almacena en esa constante un string que dice que el usuario ya existe, por último pasamos una propiedad error con esta constante (errorMsg)



Luego que enviamos estas propiedades al UserForm vamos a pasárselas por parámetros a la función.

Para manejar el estado de carga vamos a pasarle la propiedad disab

led a los input y al botón la cual evaluara si llega será true y sino será false.

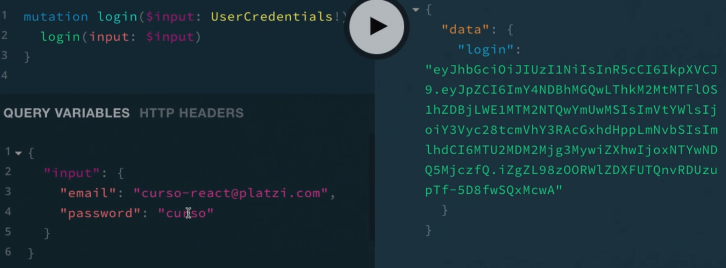
Para mostrarle al usuario que está cargando podemos hacer que se vea más opaco el formulario usando styled-components, recuerda que puedes pasarle propiedades y qué hacer cuando le llegan:

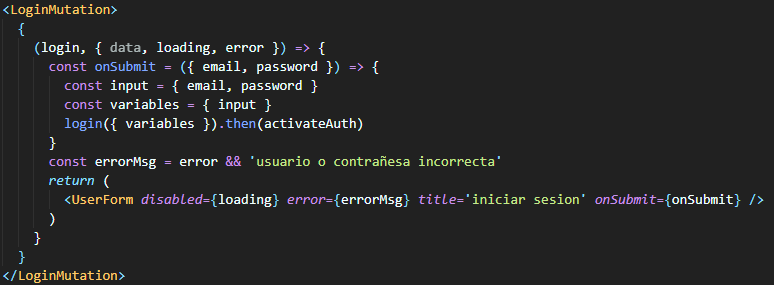
Si le llega la propiedad disabled al input y botón bajara la opacidad

Y para manejar el error, fuera del form abrimos llaves para usar javascript y evaluamos si llega el error entonces renderizara un span que mostrara el error que pasamos por propiedades el cual es un string

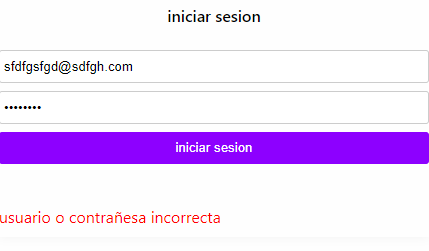
LOGIN

Para el login hacemos exactamente lo mismo, solo que vamos a cambiar la query de graphQL que ya está preparada para un usuario registrado:

Si registramos un usuario correctamente nos devolverá un token. En esta oportunidad usaremos la mutacion login que ya fue creada en el backend, si le enviamos los datos de email y password de un usuario ya registrado este nos devolverá otro token lo cual nos afirma que el usuario está registrado, si enviamos un dato equivocado nos enviada un error

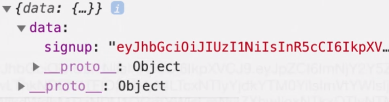
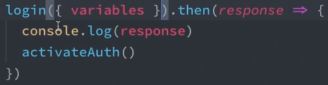
 

Si ingresamos un usuario que no esta registrado nosmostrara el errorMsg que esta vez dice que el usuario o contraseña son incorrectas:

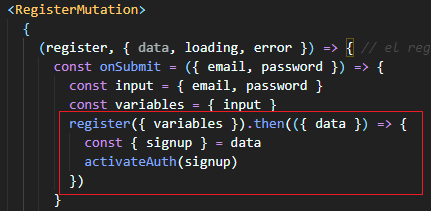


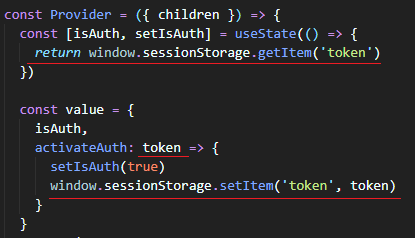
Por ahora tenemos el registro y el login de los usuarios pero al refrescar la página estamos perdiendo estos datos, ahora tenemos que buscar la manera de persistirlos para que al refrescar la página aún se queden guardados los datos y si el usuario esta logeado se mantenga logeado, de igual manera que la información del usuario registrado se mantenga en el servidor

Si realizamos un console.log de lo que nos llega de la promesa del register o del login notaremos que nos llega un objeto llamado “data” y dentro de ella tendremos la key “signup” (“login” para el caso de login) con el json web token

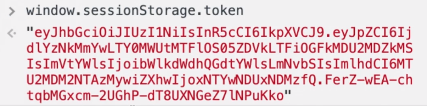


Esto significa que podemos usar este token para mantener el usuario logeado una vez inicie sesión o se registre directamente.

Para esto solo debemos cambiar el comportamiento de la promesa (.then) el cual anteriormente solo llamaba al método activateAuth y lo cambiaba de false a true, y decirle que esta vez va a recibir ese objeto data y lo que hara con él será tomar el key correspondiente, signup o login, de ese objeto data y lo meterá dentro del método activateAuth

En el context.js que es donde manejamos ese método (el cual era una función que solo seteaba el estado a true) esta vez la función recibe un parámetro que llamaremos token, seteara el estado a true y llamara a window.sessionstorage.setItem() y creara un key llamado “token” y dentro de este tendrá el token que nos llegara por parámetros desde el RegisterMutation,

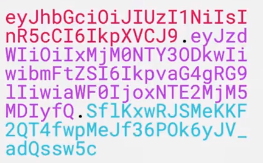
Si accedemos al window.sessionStorage.token desde la consola nos mostrara que el token si se esta almacenando, pero aun asi no lo etamos usando de manera que si refrescamos la pagina vovlemos a perder la sesión



Entonces debemos inicializar el estado con una función que retorna un window.sessionstorage.getItem() que toma el token que ya creamos y lo almacena en el estado ya seteado, de manera que almacenara el true mas el token

**Que son los JWT (json web token)?**

Un JSON Web Token (**JWT**) es un estándar abierto para crear tokens y asegurar que el envío de datos es confiable y seguro. Van a ser muy útiles para implementar la lógica de los likes pues solamente los usuarios autentificados podrán dar like.

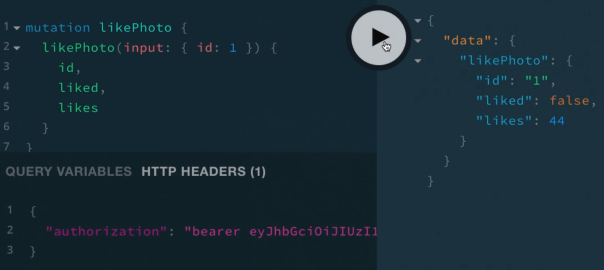
Un JWT se conforma de 3 partes:

1. Header (la parte en rojo): Es un objeto que define qué algoritmo y tipo tiene el token.
2. Payload (la parte morada): La información que almacenamos en el token.
3. Verify Signature (la parte azul): Una encriptación del header más el payload más tu llave secreta.

Para utilizar nuestro JWT necesitamos añadirlo al header authorization de las peticiones HTTP que hagamos con el texto Bearer [token].

Ya tenemos la parte donde creamos un JWT y se lo devuelve al navegador para mantener la sesión

Ahora queremos hacer una petición y enviarlo en en header, luego el servidor va a verificar si es correcto y nos devolverá la información que solicitamos

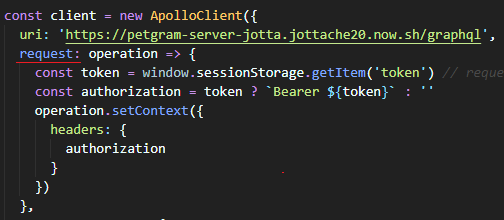
en el backend con graphQL ya esta creado el método para hacer esta acción.

La mutacion likePhoto recibe un id y devuelve el id, liked y likes

Para activar este método debemos pasarle a graphql una autorización con la palabra “bearer” seguido del token de manera que verifique si es el token correcto para devovler la información

Una vez le damos play a este código cambiara el liked a true y aumentara el número de likes. Recuerda que todo esto ya está creado y codeado en el backend con graphQL

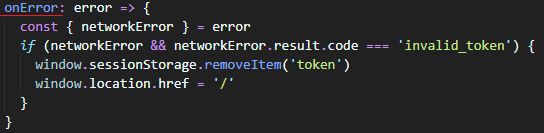
Index.js

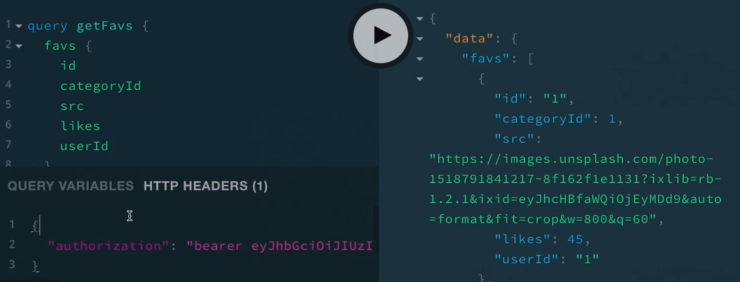
Luego en el index.js donde empieza toda nuestra app tenemos que agregarle a apolloClient otra opción, esta se llama request y se ejecuta justo antes de hacerle cualquier petición al servidor.

Entonces con esto lo que hacemos es crear una función la cual toma el token que ya esta en el sessionStorage y en una constante llamada autorization (recuerda que en graphQL debe tener este nombre) le hacemos un operador ternario que dice que si llega el token entonces devuelva un template string con la palabra bearer mas el token, sino que devuelva un string vacio

Luego en operation que es el parámetro de la función le aplicaremos un setContext para meterle el headers con su autorización y enviarla al servidor

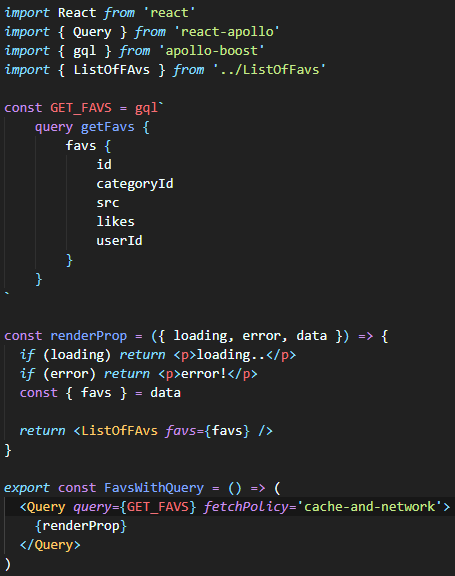
También tenemos que manejar el error ya que la sesión puede expirar.

Para esto lo que debemos hacer es crear la opción onError de apolloClient y crear una función error la cual mete en una constante el dato networkError que viene de error. Luego hacemos un if donde si llega networkError y el networkError.result.code (el cual es un dato que llega cuando hay un error y te dice el tipo de error) y este es igual a “invalid\_token” entonces va a remover el Token que está en el sessionStorage y además te mandara a la ruta home con window.location.href

**Mostrar favoritos y soulcionar fetch policy**

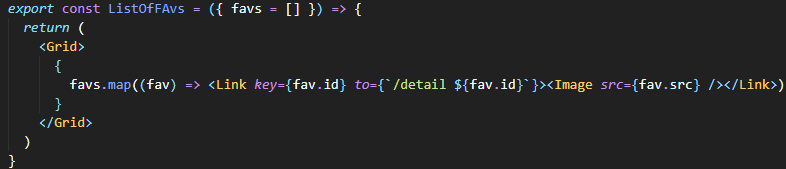


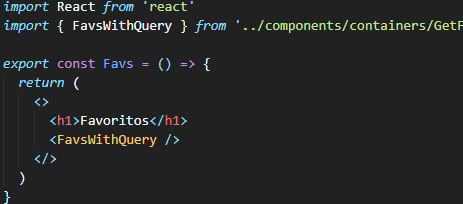
Aquí tenemos la query que nos permitirá obtener los favoritos, esta query ya está configurada para que necesite una autorización (Token) para que pueda funcionar, de lo contrario da error.

luego creamos el componente container (recuerda que en la carpeta container van los componentes que con mas lógica, en esta app los que hacen fecth con graphQL). Este componente va a realizar la query y nos devolverá ciertos dato que vamos a utilizar.

Luego creamos la renderProp que recibe los datos, loading y error.

Manejamos cada uno de ellos con condicionales y en caso de ser efectivo todo vamos a acceder al objeto favs que viene de data directamente.

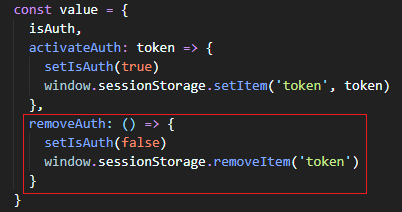
Luego retornamos un componente al que le pasamos estos datos por props (la prop se debe llamar favs por que en el componente presentacional vamos a destructurarlo con ese nombre) 

Luego que ya tenemos la render prop vamos a crear la función que tendrá el jsx, en este caso renderizara un componente Query que viene de react-apollo y a la propiedad query vamos a pasarle el query que realizamos anteriormente.

El fetchPolicy nos sirve para poder almacenar datos y hacerlo tambien offline, de manera que si le damos like a una foto y nos vamos a la pagina de favs nos aparecerá dicha foto.

Exportamos la función y la importamos en la page favs.js que es la que muestra los favoritos. Luego la renderizamos en el jsx

Cerrar sesión

para cerrar sesión tenemos que modificar el estado de auth (que esta en true para mantenerse autenticado) a false y remover el token que esta en el sessionStorage. Para esto solo tenemos que crear una función mas en el contexto.

Nos vamos a context.js y agregamos en el objeto value un nuevo key con esta función, es muy parecida a la función que usamos para iniciar sesión (la que esta arriba)

Ya en la página de User.js importamos el contexto y lo usamos con el Context.Consumer y hacemos el procedimiento para consumir el removeAuth.

Dentro del context.consumer vamos a abrir llaves para usar js y agregar la función que recibe el contexto que acabamos de crear, este retornara un botón que llama a ese contexto el cual cambia el Auth y remueve el token



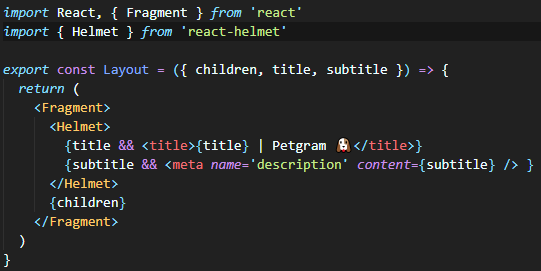
**Mejores prácticas, SEO y recomendaciones**

En este modulo vamos a ver como podemos añadir temas de SEO como podemos añadir la performance de nuestra app, veremos react.lazy, react.memo, suspense, las proptypes, PWA y testing con cypress.

**React Helmet**

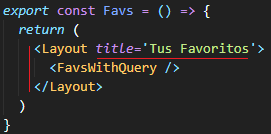
Las SPA no necesariamente tienen mal SEO como se hace pensar. React Helmet nos va a ayudar a manejar el SEO de nuestra aplicación

Primero instalamos React-helmet con npm i react-helmet la cual será una dependencia de desarrollo

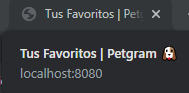
Creamos un componente que llamaremos layout, este componente tendrá nuestro react helmet.

Lo que hace react helmet es darnos un componente que podemos usar para mejorar la parte del head del html de cada página, la cual no podemos modificar con react normalmente.

Importamos el componente helmet y a este le haremos las siguientes codiciones, si llega un title entonces va a colocar el title que le llega por props, al igual que el meta, etc.

Tambien le llegaran los children por props ya que este componente va a encerrar a todos los hijos en donde lo coloquemos.

En la página favs.js vamos a encerrar todo dentro de el componente layout que importaremos, y a este le vamos a pasar las propiedades que queramos cambiarles. Por ejemplo el title



**React.memo()**

¡Los componentes solo harán *render* si sus *props* han cambiado! Normalmente, todos los componentes de React en nuestro árbol pasarán por un render cuando se realicen cambios. Con PureComponent y React.memo(), podemos tener solo algunos componentes renderizados.





En este ejemplo en el componente Home lo que hacemos es cambiar el nombre del componente como ya lo teníamos y quitarle el export, luego al final del componente vamos a exportarlo con el nombre que tenía normalmente pero usaremos React.memo.

Lo que hace es evitar el re-renderizado del componente si este recibe las mismas props, incluso podemos pasarle una función que compara si tiene las mismas props o han cambiado para que funcione mejor.

React.memo es un método que solo debemos usar cuando un compoente es demasiado grande y queremos evitar que tarde demasiado en cargar por re-renderizados, si nuestro componente se renderiza muy rápido no será necesario usarlo.

Para revisar la performance de nuestros componentes podemos correr nuestra app en modo de producción con el siguiente script



Este script nos permitirá abrir nuestra app en modo de producción para ver como se comporta, vizualizaremos su comportamiento con las react dev tool o tambien con la opción de “performance” que nos saldrá en el inspector cuando la app corra en producción.

**React.lazy() y componente Suspense**

Suspense es un componente de React que nos va a permitir suspender algo cuando está en modo lazy(); y lazy(). El cual nos va a permitir importar un componente que no será cargado hasta que este sea llamado. De esta forma mejoraremos el tiempo de carga de nuestra aplicación enormemente.

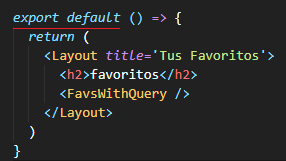
Lo primero que debemos hacer es importar Suspense de react

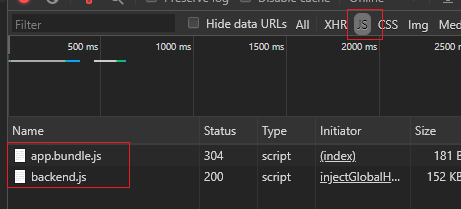


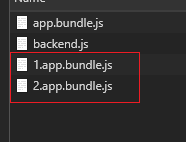
El cual es un componente que nos permite colocar en suspensión las pages que queramos que carguen solo cuando el usuario lo necesite o solicite, este componente debe encerrar toda la app y debe llevar una prop, esta prop es fallback y debe ser un componente que colocara en su lugar mientras se suspende (creo yo)

Luego debemos cambiar los imports que queremos que sean suspendidos por el método de react.lazy para hacer imports dinámicos.

Este import debe etar metido en una constante que tendrá el nombre del componente, lo igualamos a React.lazy el cual lleva como parámetro una función y esta lo que hace es el import del componente.

cada componente que coloquemos en suspensión debemos cambiarle la manera en como lo exportamos. Ahora ya no será export nombrado, sino export default únicamente, no es necesario ni colocarle nombre.

cuando corremos la app notamos que nos carga el bundle.js que es donde esta nuestra app, pero cuando le damos al botón de favoritos el cual fue el que le colocamos el React.lazy entonces nos cargara ese componente aparte

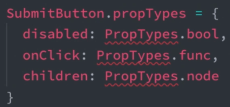
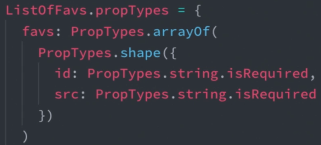


En este punto, es normal que tengas diferentes archivos javascript dentro de la carpeta. A esto se le llaman “chunks” o, dicho de otra forma, son trozos de tu bundle. Estos “chunks” son descargados de forma asíncrona cuando lo necesita la aplicación. ¿Y de dónde salen? Pues por cada import dinámico que haya en tu código, tendrás uno de estos chunks. Por ejemplo, uno sería para el Intersection Observer polyfill, otro para cada una de las rutas que estamos cargando de forma asíncrona

**Usando PropTypes para validar las props**

Las PropTypes serán un validador del tipo de datos que estamos recibiendo como *props* en nuestros componentes, el cual nos permitirá a que sea exclusivamente ese tipo de datos.



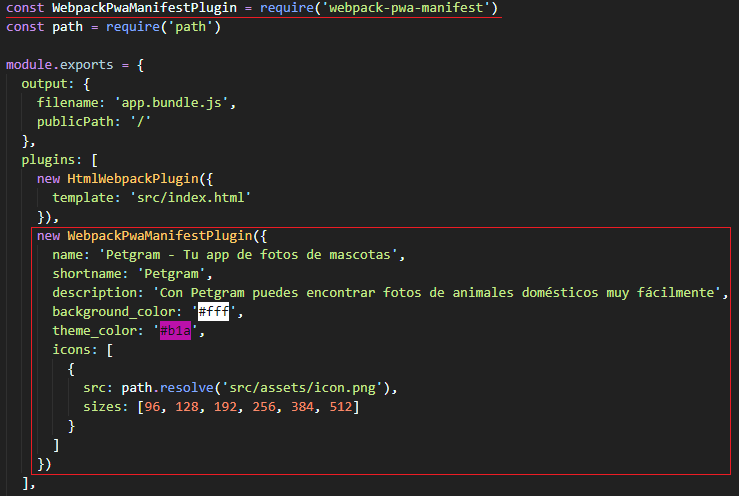
 

Debemos usar el nombre del componente que queremos validarle los datos y agregarle propTypes (la P va en minuscula) el cual será un objeto que contiene como keys las propiedades que le llegan al componente y como valor tendrá Proptypes (la P en mayúscula) seguido del tipo de dato. Puede ser bool para boleanos, func para funciones entre otras. Existe un tipo de valor que se llama “node” que es para cualquier tipo de elemento que pueda renderizarse, este se usa para cuando llegan children por props.

Si le agregamos isRequired nos pedirá que una prop llegue obligatoriamente, sino nos informara que no está llegando dicha prop

**PWA: generando el manifest**

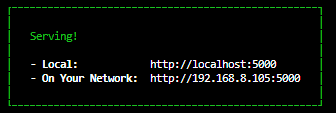
Usaremos webpack-pwa-manifest para crear nuestro manifest.json y que nuestra aplicación pueda ser compatible con todos los requerimientos de una PWA. Lo instalamos con npm install webpack-pwa-manifest –D

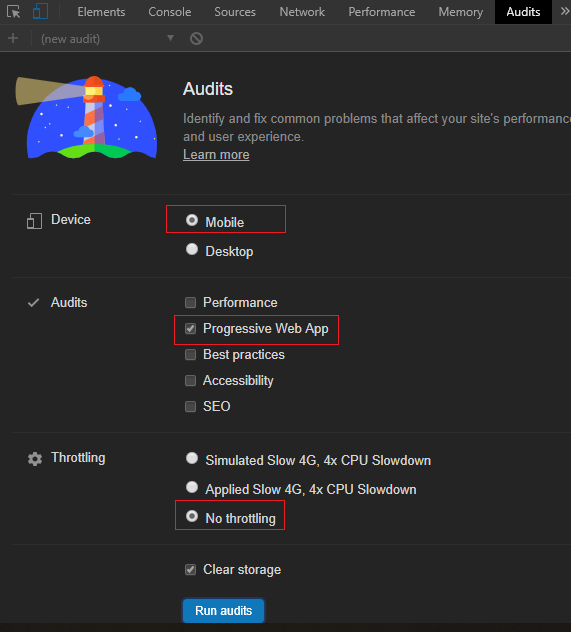
Una vez instalado vamos a crear el manifest, lo requerimos en webpack y lo instanciamos pasándole un objeto con las propiedades que necesita para crearse. El name, shortname, description, background\_color (para el color de background del manifest), theme\_color(para el color del tema en el navegador) icon el cual será un array que dentro dentra un objeto con la ruta del icono y los tamaños que nos va a crear.

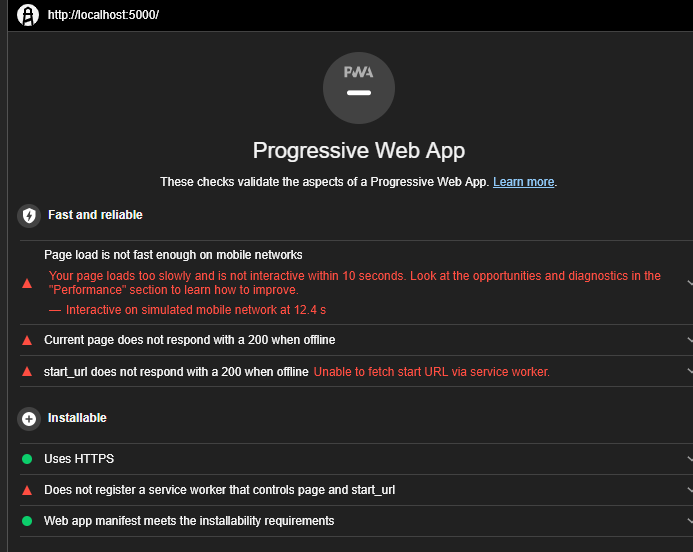
estos que aparecen aquí son los que mas se usan generalmente

Luego corremos el comando que nos crea la app en producción para probarla y la abrimos en el puerto que dice.



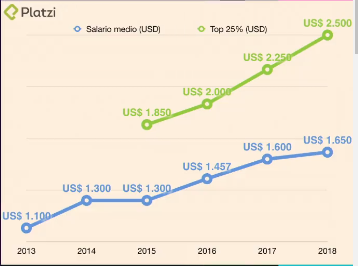


Abrimos la página en el puerto que nos indica y nos vamos al inspector en la opción audits, en esta vamos a seleccionar únicamente las que están marcadas para revisar que tanto nos falta para tener una PWA



Como desarrollar tu crecimiento profesional en tecnología, la carrera de mas crecimiento en el mundo actualmente con la demanda mas fuerte del mercado y el sueldo mas alto

En el 2013 el sueldo promedio



1000$ a 3500$ en latino america

Como salir en la grafica

3 posibles cargos,

Colaborador individual (la mayoría que trabajan en una empresa) IC



Manager o ejecutivo (gente bajo tu cargo)

Fundador o emprendedor Founder



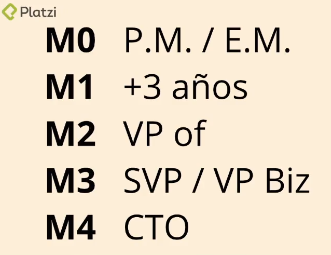
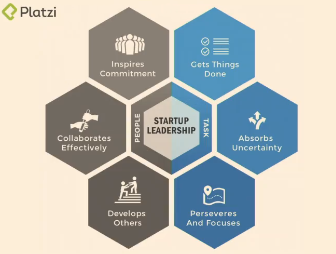
Cosas que se necesitan para trinufar en esta industria:

Aprender inglés. (Leer pedro galban)

Tener una marca personal profesional para mostrar skills logros y progreso así sea proyectos personales

Hablar en público (muy importante)

Fundamentos sólidos de lo que sabemos, entender el porqué de las cosas

Si lo que los ofrecen para crear o desarrollar no te produce esa sensación de emoción mas que de dinero la respuesta que deberías decir es NO por que frenara tu crecimiento personal profesional

