Creación de una aplicación REACT

A través de la ejecución del comando desde consola, ubicado en una carpeta determinada

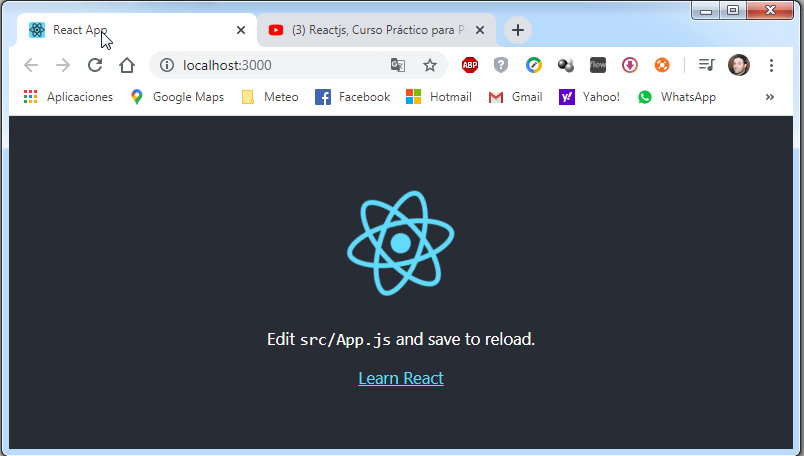
npx create-react-app myfirsapp

lanzará la creación de un proyecto REACT en una carpeta que tendrá el mismo nombre que le dia a la aplicación en el comando.

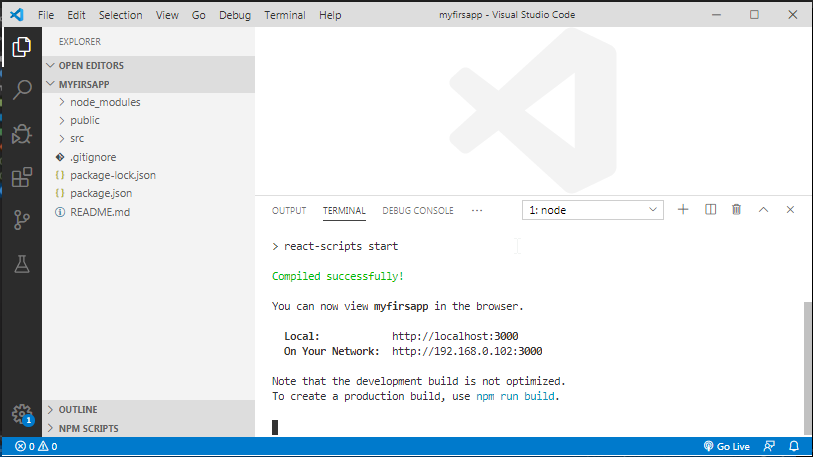


Una vez finalizado me muestra una serie de comandos que podre ejecutar para ir trabajando, el primero inicia un servidor que hace que cada cambio que efectué en el código se detecte y se vuelva a compilar el proyecto, otro para generar los archivos de la aplicación para producción, etc.

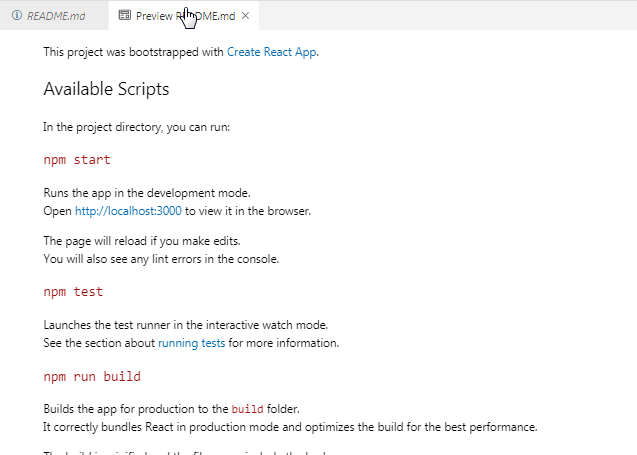
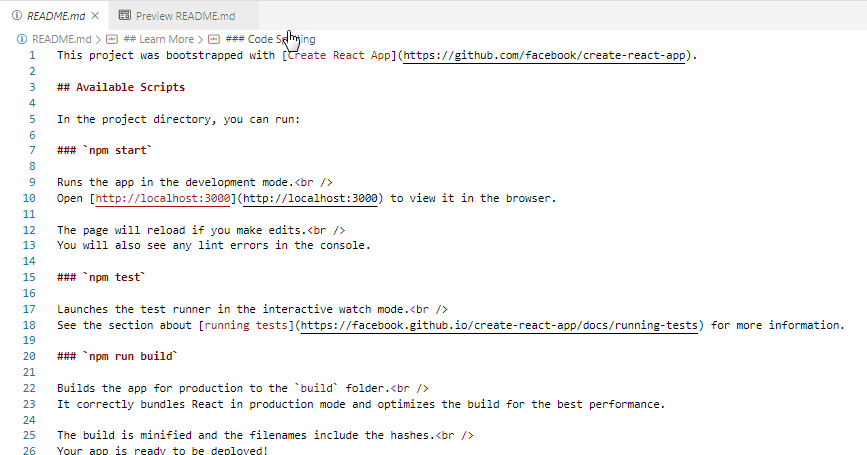
Puedo utilizar VSCode para abrir el proyecto, directamente lo puedo hacer arrastrando la carpeta sobre el VSCode y lo abre. Y ya puedo abriendo una terminal ejecutar el comando npm start, que inicia la aplicación inicial básica, un servidor escuchando en el puerto 3000.



El proceso de instalación crea una estructura de carpetas y archivos y eso es lo que vamos a analizar ahora para tratar de entender cuál es la estructura de un proyecto REACT.



En primer lugar los archivos en la raíz del proyecto el readme.md tiene información del proyecto que podemos ampliar y sirve para documentar lo que hace el proyecto y todo lo que se crea conveniente documentar para ayudar a otros desarrolladores que puedan estar interesados en analizarlo. Este archivo es de texto puro y con un formato Mark Down, de manera que si se lo abre con F1 (o Shift+P)-> MarkDown: Open preview



El archivo package.json contiene los datos dl proyecto, las dependencias y algunos scripts. Las dependencias son los modulos que el poryecto necesita para funcionar. En este caso inicialemente se ven tres dependencias

**React** 🡪 que es el núcleo en sí de React, la biblioteca que nos permite diseñar las interfaces.

**React\_Dom** 🡪 que funciona de manera conjunta con la anterior para desarrollar en este caso una aplicación web. Pero esta la puedo reemplazar por otra, que en conjunto con React nos permitirá el desarrollo de aplicaciones móviles, o quizá por otra para desarrollar aplicaciones de escritorio Por esta razón se las separa.

**React\_Script** 🡪 tiene los módulos que React necesita para funcionar, configurados ya que React se complementa con otras tecnologías, esas tecnologías necesitan configurarse, generar archivos y demás. Y esas configuraciones esta ‘escondidas en estos módulos de aquí.

Los scripts en el json son comandos que podemos ejecutar desde consola como por ejemplo en **npm run start** para arrancar la aplicación. El build, por ejemplo, será el que finalmente convertirá todo a código JS.

Otras partes del json por ejemplo indica especificaciones para las listas de navegadores en que se podrá dar soporte.

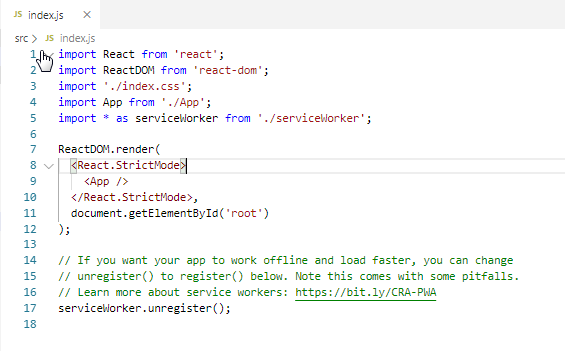
El archivo package-lock.json, este tiene información de que módulos y que versiones de los módulos tiene el proyecto. Este archivo generalmente no debe tocarse.

El .gitignore… posee la lista de archivos que se ignoran a la hora de subir al proyecto en git.

La carpeta /public, contiene el tamplate o el archivo que nos va a permitir montar la aplicación de React. En resumen es el archivo html de toda la vida, al que le agregamos css y JS. Dicho archivo luce básico, incluso del bloque <body></body>, que no contiene nada más que um bloque <div id=”root”></div> y esto es porque en React todo lo que vemos en pantalla no es generado con html sino que es generado con código JS. React selecciona este div y lo comienza a alterar con JS.

También para el caso de no tener el JS activo en el browser… hay un mensaje que pone de sobre aviso al usuario en el <noscript></noscript>

La carpeta /src es donde está toda la aplicación. El archivo index.js es el archivo inicial. Si miramos su contenido



básicamente están

* los import a las librerias **React** y **React\_dom** (específica para desarrollo web).
* un import a un archivo index.css que no es mas que un archivo de estilos básico.
* un import a una librería serviceWorker relacionado a pregressive web app (PWA) que es en la manera en que se pueden crear aplicaciones en estos días (enviar notificaciones, tener acceso a los archivos, ejecutar cosas en segundo plano, etc). Puede quitarse si no se va a utilizar nada que tenga que ver con desarrollo de app web progresivas.
* El import de un archivo ./App.js
* la línea

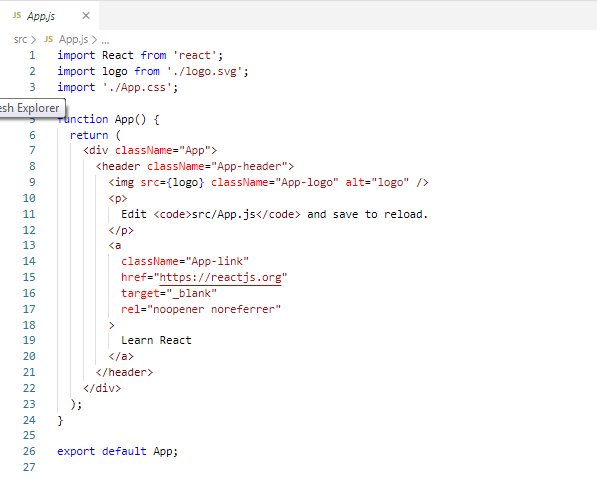
ReactDOM.render(<React.StrictMode> <App /> </React.StrictMode>,document.getElementById('root'));

ReactDOM dice: voy a utilizar React para pintar algo por pantalla, voy a pintar un componente </app> que es el que contiene la imagen, el texto y el enlace que muestra la pantalla inicial (que se mostro más arriba), y le indico además en que elemento del DOM quiero pintar, es decir, en el elemento del html cuyo id es ‘root’ (que vimos era el único div en el body del index.html).

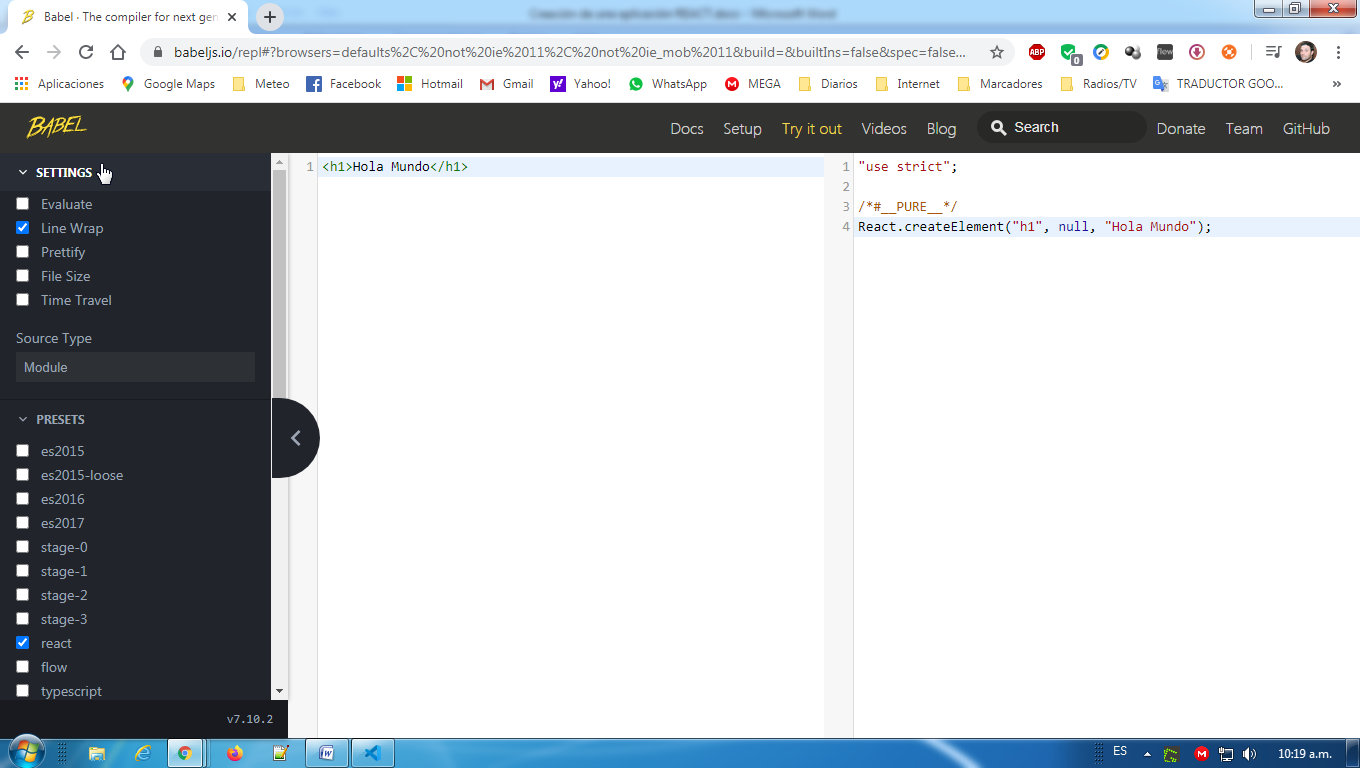
El componente </App> esta en el archivo App.js que se importo.

Si miramos el código del archivo App.js vemos algo que parece código HTML pero no es HTML es JS, más precisamente JSX, es muy similar al HTML pero no lo es.

Mirándolo detenidamente allí está el logo que se ve en pantalla, los párrafos y el link representado como si fuese un html pero, por ejemplo, si fuese html seria <div class=’App’> y no <div className=’App’>, y así se pueden notar algunos otrs detalles que lo diferencian, pero básicamente se parece y resulta fácil de comprender



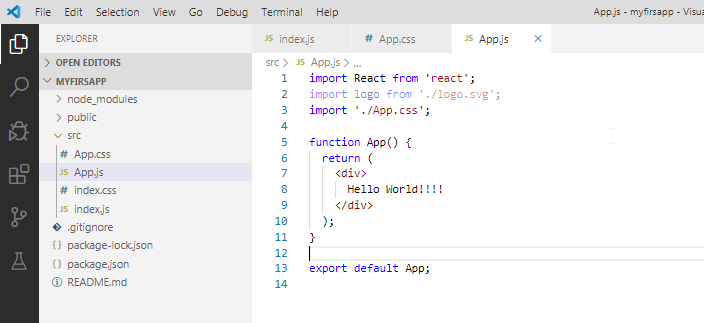
Como se menciono en React el html se transforma en JS. Una forma de ver como se React traduce html a JS puro es haciéndolo por Babel REPL (<https://babeljs.io/repl>) (min 42 del video)



si bien lo que muestra el código de App.js es otra cosa más parecida a HTML, y lo que muestra Babel REPL como traducción del HTMl es React puro (también JS)… sin embargo se ve a las claras que JSX es mas legible y React tenemos la alternativa de escribir utilizar en el propio código la sintaxis parecida a HTML, o sea JSX.

Explicado esto, podemos avanzar un poco más.

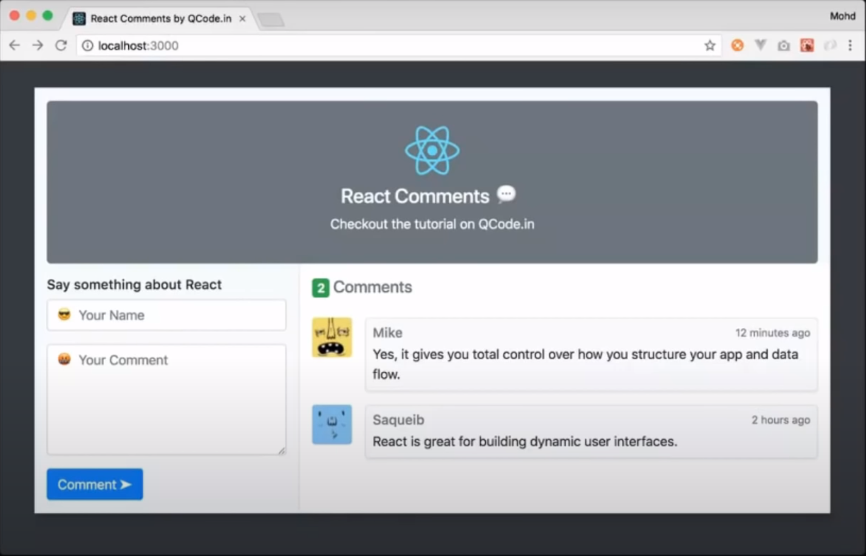
Eliminaremos algunas cosas del proyecto inicial, quitamos algunos archivos como los de test, eliminamos los contenids de los .css y del app.js solo dejamos un bloque dic con Hola Mundo.

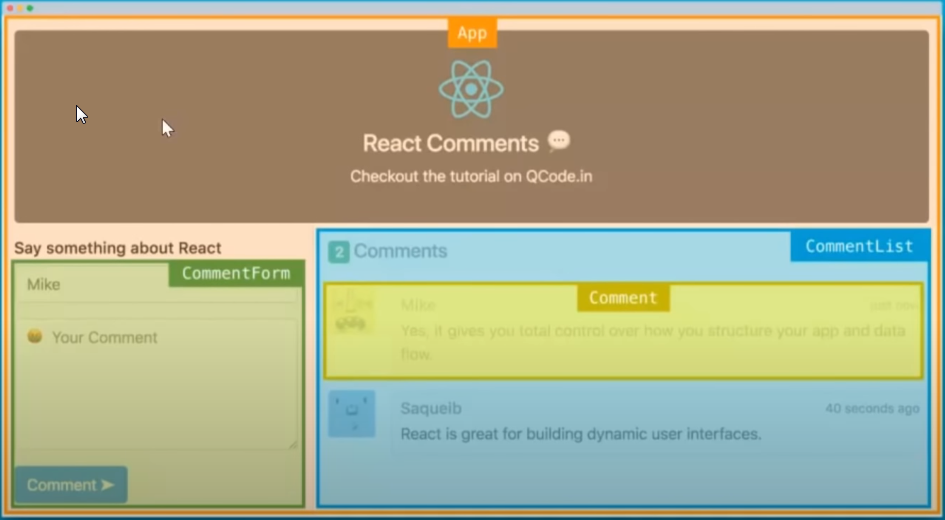


# Componentes

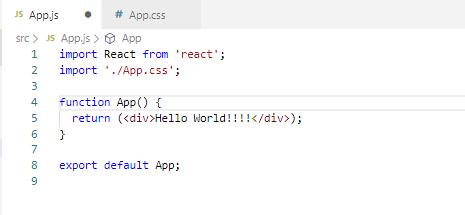
En principio vamos a tratar de entender el concepto de ‘**componente**’. Mirando una página web sencilla como la de la imagen. Vemos que tiene un bloque con una imagen, un titulo y un párrafo. Más abajo se detecta claramente sobre la izquierda un formulario ya que tiene un cuadro de edición, un área de texto y un botón y sobre la derecha una sección donde se listan los comentarios bajo un titulo. Además cada comentario en la lista muestra una imagen un párrafo de comentario el nombre del autor y la fecha de publicación del comentario.

Entonces podemos pensar que la pagina se puede dividir en tres partes, sección 1, sección 2 y sección 3 y podemos representarlo como en la siguiente figura a la imagen de la página, donde se ve un componente global App, un commentForm, un commentList y un comment. Como se ve el principal contiene al commntForm y al commentList y a su vez este último contienen al comment. De esta manera es como tenemos que comenzar a ver las aplicaciones en React, como subdividir lñas aplicaciones. Se basa en dividir algo grande en partes más pequeñas dependiendo en lo que se necesite.



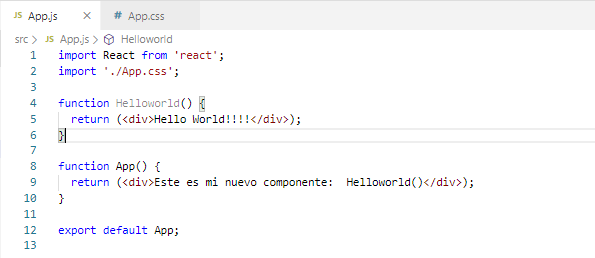


Bien… ahora vamos a ver como escribir un nuevo componente y como llamarlo desde App.js



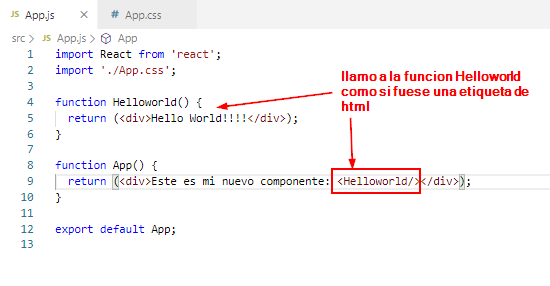
En principio vamos a crear una función Helloworld() que se corresponderá con el nuevo componente, gralmente se estila a los componentes llamarlos con su primera letra en mayúscula.

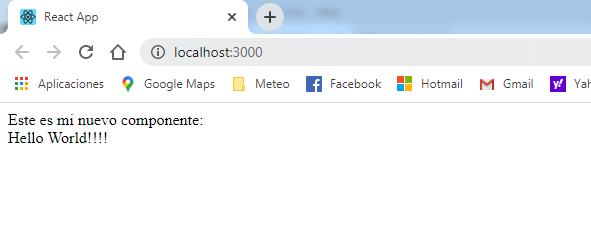
En JS las funciones la llamaríamos dentro del código como Helloworld()



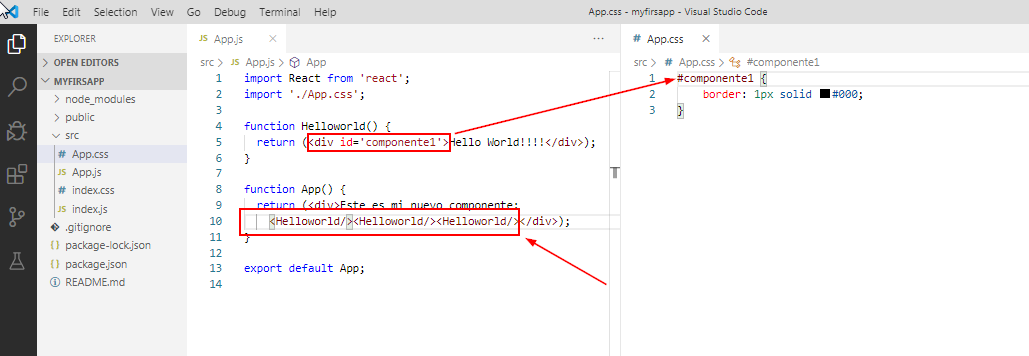
Pero en este caso esta como dentro de una notación en HTML (JXS) para llamarla lo hacemos de la siguiente manera <Helloworld/>.

Como se ve en la figura de más abajo y después se muestra como se vería desde el browser. También vamos a ver como agregar estilo, lo hacemos poniéndole un borde al texto, para hacer esto lo haríamos desde un .css, pero en este caso lo hacemos desde el archivo App.css. Entonces tendríamos primero que darle un id al div del nuevo componente como para referenciarlo desde el App.css

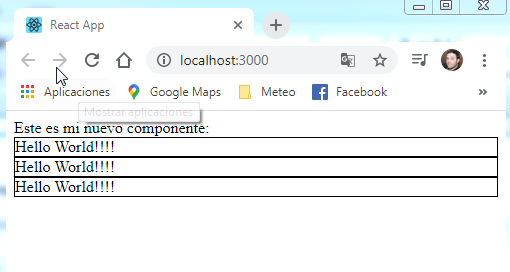




además de darle estilo vamos a agregar dos veces más el componente

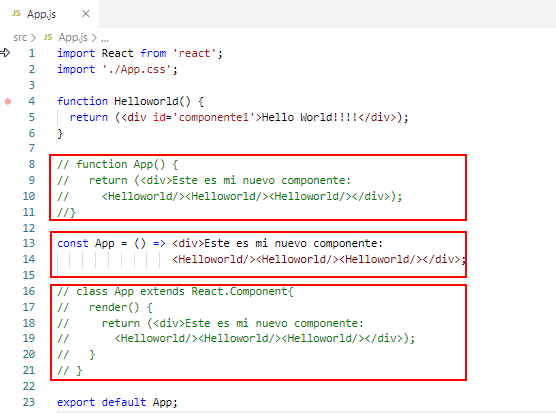


el resultado será



Esto último muestra cómo puedo reutilizar el componente recientemente definido.

Otra cosa que se puede destacar es que la función App pudo haberse escrito con la definición de función flecha o también puede definirse como una clase





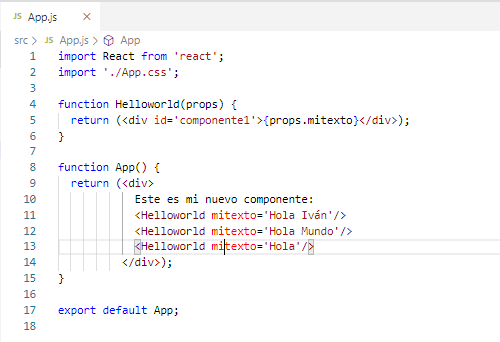
Para hacerlo como una clase es necesario hacerla extender desde React.Component y utilizar el método Render.

# Propiedades

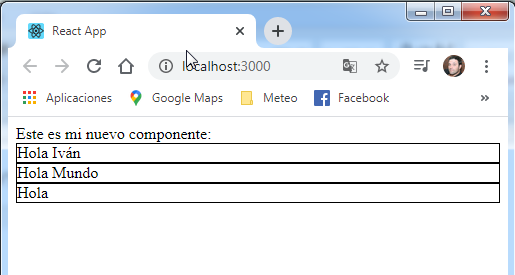
Veamos ahora la necesidad de poder reutilizar un componente pero con la posibilidad de realizar algún cambio en su contenido, por ejemplo supongamos que quiero llamar al componente Hellworld pero que muestre un texto distinto. Para esto vamos a ver el concepto Props (propiedades o properties).

Lo más fácil es verlo con un ejemplo. En el tag <Helloworld/> vamos a agregar una propiedad con el nombre que se me ocurra por ejemplo mitexto y le asigno un texto determinado. Por otro lado en la función Helloworld definimos un parámetro para la función al que le damos un nombre props (puede ser cualquier otro). Y en el código de la función hacemos referencia a él entre {} (lo que está entre paréntesis se interpreta) pero teniendo en cuenta que props es un objeto JS es decir

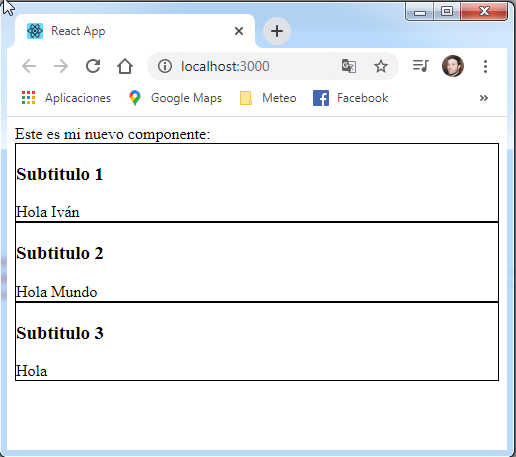
props 🡪 {mitexto:”Hola Iván”}



y vemos que ahora el contenido de cada componente ahora difiere. Fue necesario también agregarle la propiedad a cada uno de los tres ya que sino, no muestra texto.



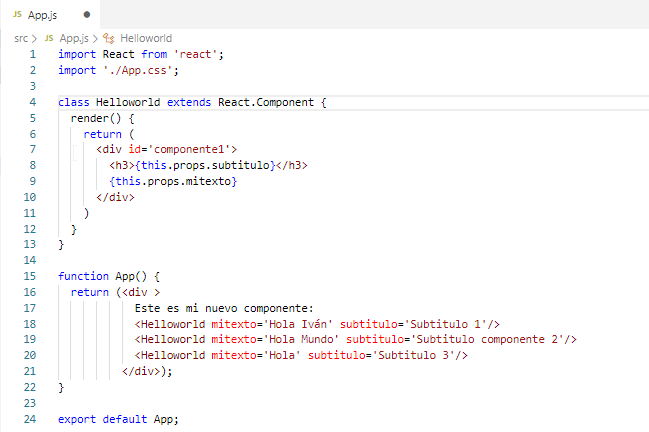
hasta el momento, este es un ejemplo sencillo pero de aquí en adelante podríamos pensar en más cambios de contenido manteniendo el diseño del componente.

# Estado

Se conoce como ‘estado’ de un componente a los datos que pueda contener internamente. Para ver un ejemplo de datos internos en un componente será mejor para comprenderlo viendo la definición del componente no como funcion sino como una clase.

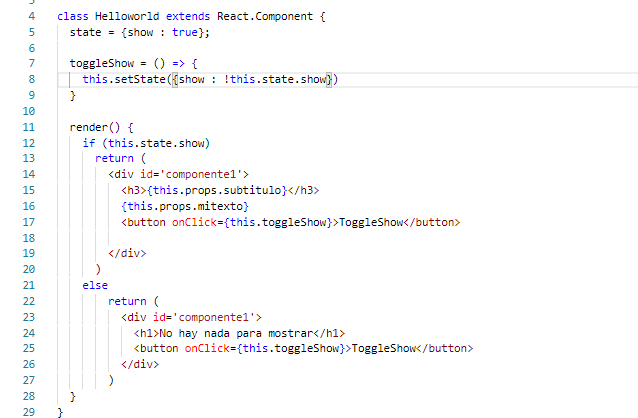
Al estar el componente definido como una clase y no como una función, props ya no es un parámetro de la función sino que es una propiedad de la clase por lo tanto tengo que referirme a ella con this, sino dará error

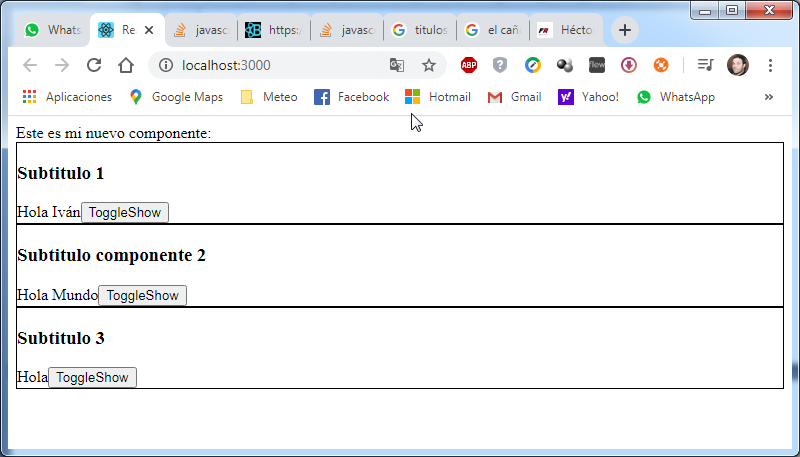


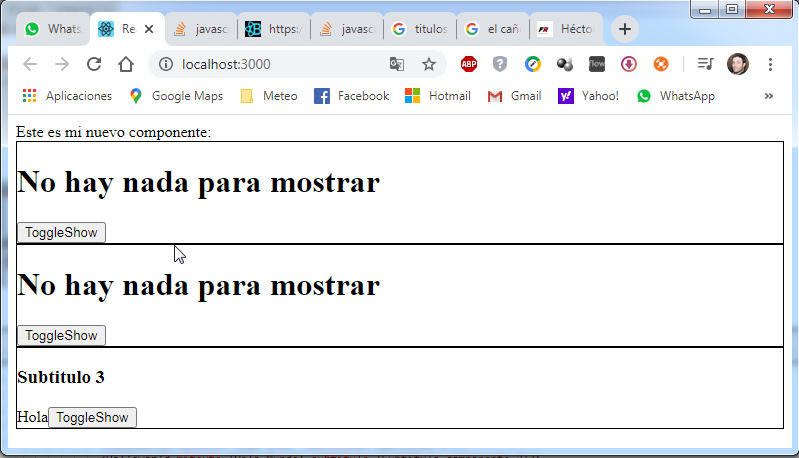
Le vamos a agregar a la clase una propiedad state, que es un objeto y lo que hace este objeto es almacenar datos de este componente, supongamos que le agregamos una propiedad show que nos permite mostrar u ocultar el componente y le damos un valor inicial true (visible). Y lo que finalmente buscamos es que nuestro componente haga algo de acuerdo a este valor.

En un primer momento se hacen cambios por código del valor, y luego se agrega un botón, de manera que con el evento onClick del mismo se pueda cambiar el estado y poder mostrar como con la acción del usuario se producen los cambios.

Lo s cambios se hacen directamente sobre la clase n el método render() de acuerdo al estado de show se muestra un html u otro, con diferentes mensajes, ambos tienen un botón en cuyo evento onClick que llama a una función toggleShow() que lo único que hace es pasar el valor de true a false







A parte del curso la dejo guardada en el archivo src\_primera\_parte\_curso.rar. Ya que vamos a eliminar el contenido de los archivos de 7src para arrancar con algo nuevo

# Task APP

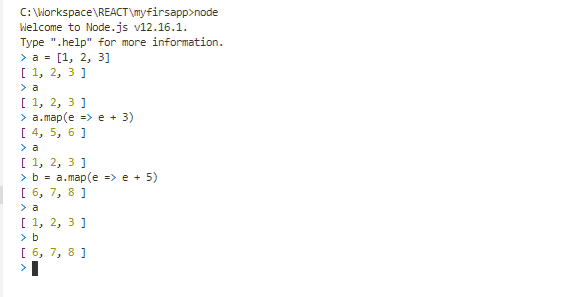
Vamos ahora a empezar con una aplicación desde cero. En primer lugar lo que vamos a necesitar son datos de prueba y para obtenerlos existen dos formas, una seria obtenerlos desde un servidor externo, un back-end pero no va a ser la forma que voy a hacerlo ahora pues para esto tendríamos que escribir el código de dicha aplicación web (en PHP, Ruby, java o cualquier otro lenguaje de bak-end), la otra es simular proveerlos creando esos datos en la misma carpeta /src. Para esto vamos a crear una carpeta /samples y dentro de ella un archivo task.json, un arreglo de objetos json tareas que tienen algunos datos básicos de lo que podría ser una tarea como id, titulo, descripción y realizada. Con esto vamos a escribir una aplicación que trabaje con tareas, que podemos listarlas, agregarlas eliminarlas, modificarlas, etc. La aplicación App no la vamos a escribir no como una función esta vez, sino como una clase.

Para poder ver este archivo desde la aplicación primero tenemos que importar ‘./simple/task.json’ desde App.js en una constante, por ejemplo tasks.

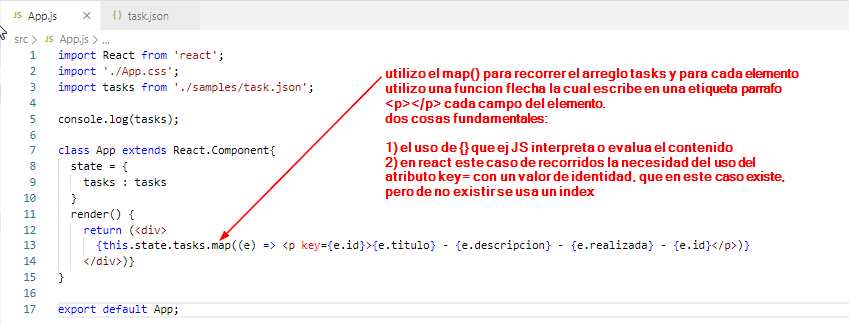
Para tomar el arreglo tasks en la clase App, lo que tenemos que hacer es: recordando el concepto de estado, entonces, definir en state una propiedad tasks que contendrá el arreglo tasks. Luego al momento de renderizar los datos a la pagina, recorremos ese arreglo de forma particular.

Para poder mostrar los datos en la pagina, vamos a tener que escribir código en el método render() de la clase App. En principio recorrer el arreglo de tareas, pero no la vamos a recorrer con una sentencia cíclica como un bucle for, por ejemplo, sino con el método map() relacionado a los arreglos, esto es porque en React esto se hace utilizando iteradores que le pertenecen al propio arreglo y la ranzón de esto, es porque React tiene un enfoque de programación funcional.

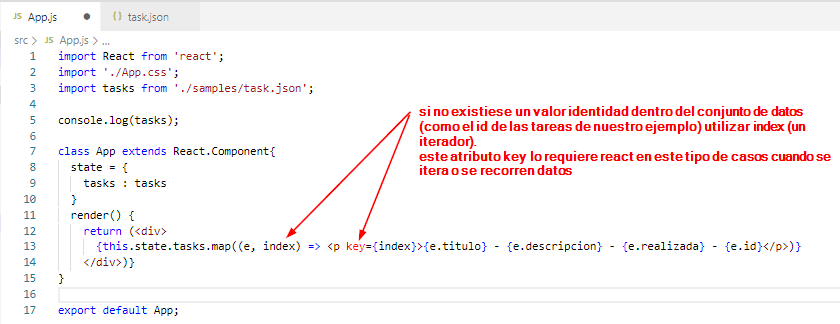
Hay que tener en claro que el método map() me retorna otro arreglo, cualquier operación que realice sobre los elementos del arreglo original dentro del map(), no afectan al arreglo original, sino que map() retorna otro arreglo. Veamos un ejemplo desde consola



ahora veamos el código de la clase App



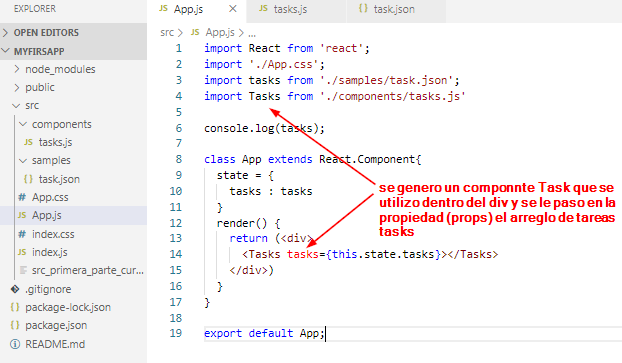
en este tipo de casos cuando se recorren elementos en un arreglo o en una lista, React requiere una clave por cada nodo de la lista como se menciono si dentro de los datos existe algún dato que identifique unívocamente a los elements puede utilizarse ese dato como valor para el atributo key para la etiqueta <p>, en otro caso suele usarse un iterador como vemos abajo.

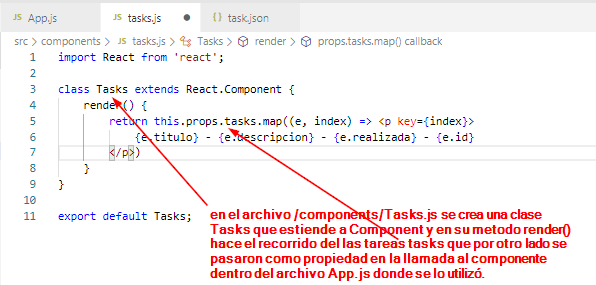


De todas maneras vamos a intentar hacer esta iteración aparte, designándolo a otro componente.

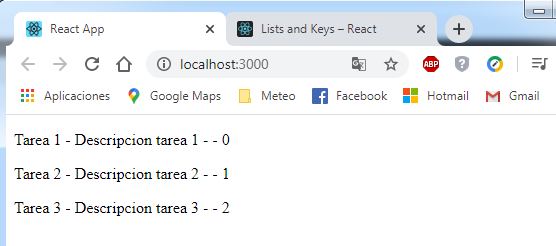
Para esto vamos a crear una nueva carpeta dentro de /src, llamada /components cuyo propósito será el de contener diferentes componentes de nuestra aplicación. Entonces allí vamos a crear un componente tareas para lo cual crearnos un archivo tasks.js e importamos en el React y también Component. Definimos la clase Tasks que extiende a Component y esa case tasks la exportamos. Por otro lado en App.ls importamos la clase Tasks para utilizar el comoponente <Tasks> (recordar que los componente siempre se definen con la primer letra en mayúsculas)

Por otro lado la lista de tareas tasks del estado (state) sigue estando definido en la clase App.js, y se lo pasa como propiedad (props) al componente Tasks para que allí lo procese.



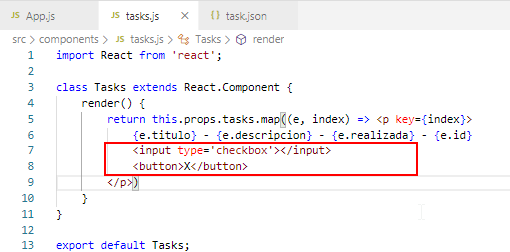


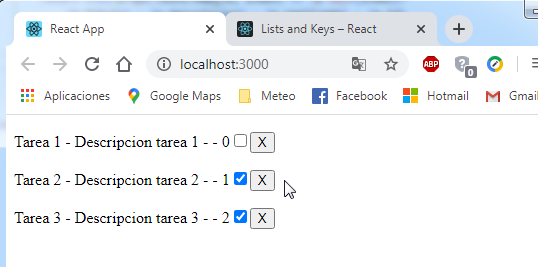
Seguirá mostrando las cosas de la misma manera pero sacamos el componente Tasks afuera.



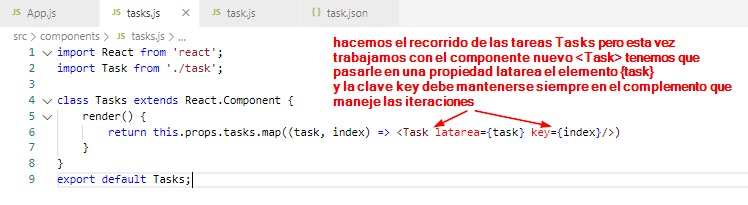
Ahora, vamos a agregar a tarea un elemento <input> del tipo checkbox que me permitirá cambiar el valor del estado ‘relizada’ a true o false y también agregaremos un botón que me permitirá eliminar la tarea.

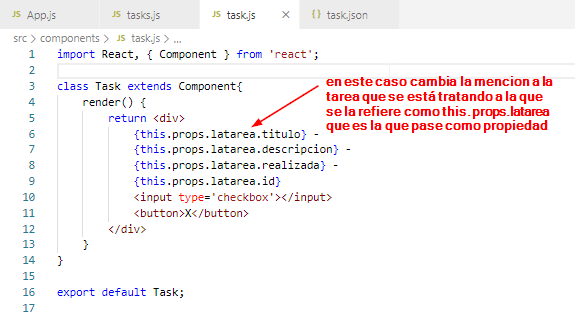
Entonces en el componente Tasks dentro de la etiqueta <p> agregamos estas dos nuevas etiquetas



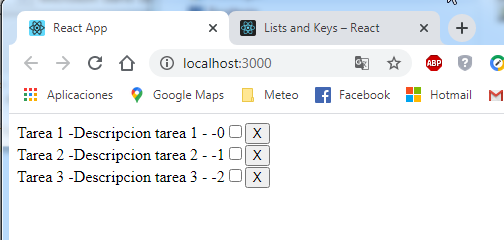


Mas adelante le iremos agregando el comportamiento. Primero vamos a factorizar un paso más, vamos a crear otro componente, esta vez Task (como unidad) ya que Tasks era una lista de tareas. Volvemos a hacer lo mismo creamos un archivo Task definimos la clase Task, la exportamos, transladamos la parte interna del tratamiento de cada tarea de la clase Tasks a la clase Task

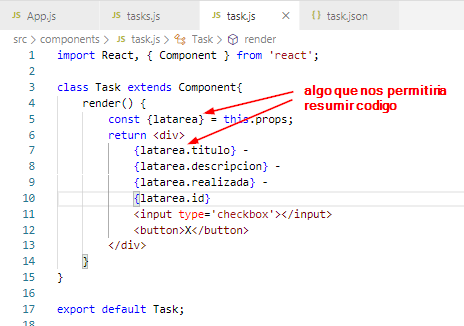




Mientras en la página seguimos viendo lo mismo



Una notación que permitiría reducir un poco el código o hacerlo mas legible



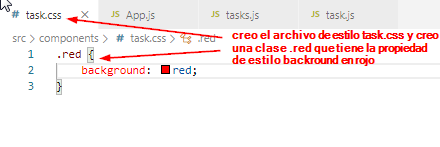
Hasta este punto hemos visto como dividir una aplicación en varios componentes y no solo eso sino que cada uno de los componentes este en un archivo diferente.

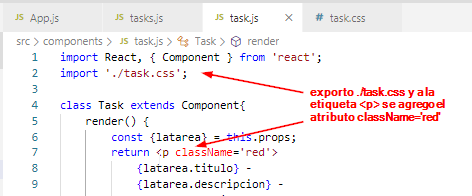
# Estilos

Ahora vamos a ver como agregar los estilos a los componentes. Gralmente para cada componente creamos un archivo de estilos y allí definimos diferentes clases, luego se importa este archivo .css desde el archivo js del componente y se utilizan las clases defiendas para las diferentes etiquetas.

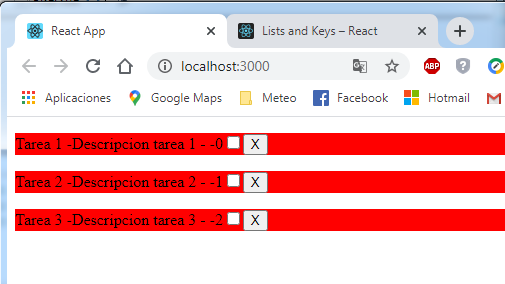
React cambia los nombres de algunos atributos de HTML (por ejemplo el atributo class por className) y lo hace por una razón de funcionamiento interno, generalmente suele avisarnos al momento de compilar. Pero son cosas que debemos tener en cuenta.

La otra forma de establecer los estilos es hacerlo en línea, es decir, en el mismo código JS. Aunque no es HTML, es JSX, la forma de hacerlo es medio parecida, a la etiqueta le agrego la atributo style y allí le doy valor a las diferentes propiedades de estilo

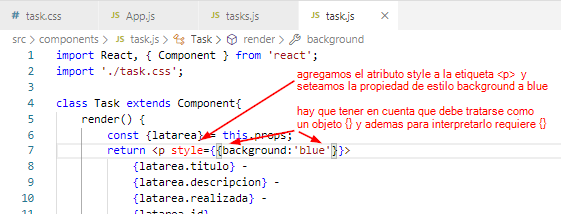


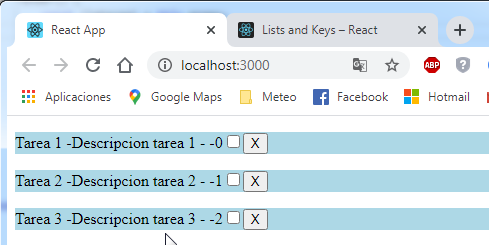


el resultado a la salida

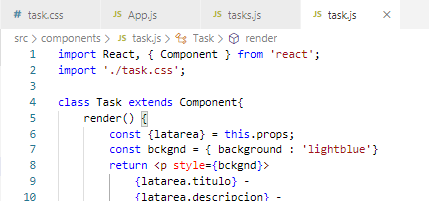


para hacerlo de la manera online, habría que tener en cuenta de el valor no se lo puedo pasar como string como se hace en un html. JSX requiere se los pase como un objeto es decir {propiedad : valor} y a su vez para que lo interprete hay que ponerlo entre {} por lo cual queda encerrado entre dos llaves

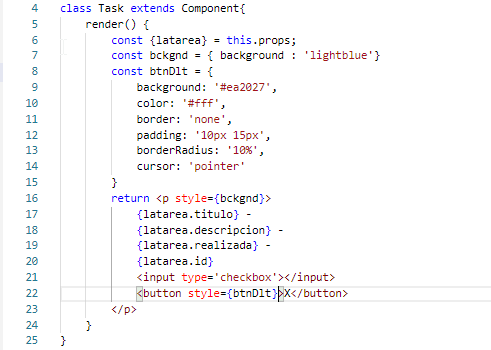
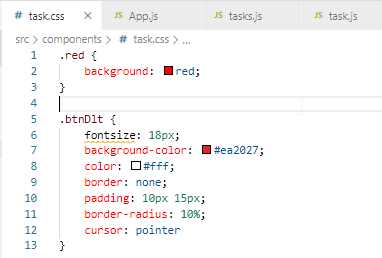




también pars evitar las dos llaves y quizá hacerlo mas legible, podríamos haber creado una constante bkgnd y asignado el objeto {backround: ‘lightblue’} y directamente en el atributo style haber puesto style={bckgnd}



Supongamos que le damos estilo al botón por un lado desde el css creando la clase btnDlt y por otro lado definiendo de manera online el atributo style dando los valores en const btnDlt

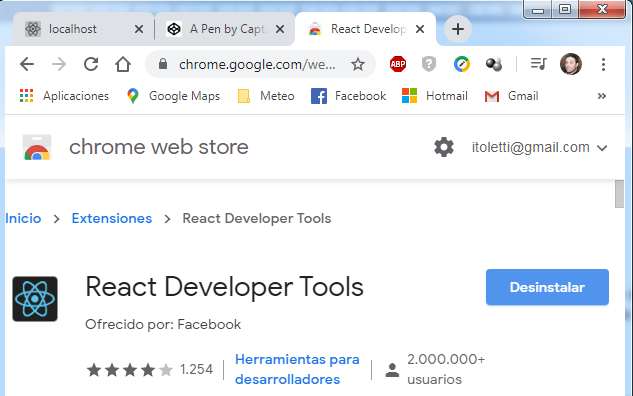
se obtendria el mismo resultado, aunque la diferencia de los nombres de las propiedades para el atributo style de JSX para utilizarlo de manera online, difieren de las mismas propiedades utilizadas en el .css para una clase denominada btnDlt.

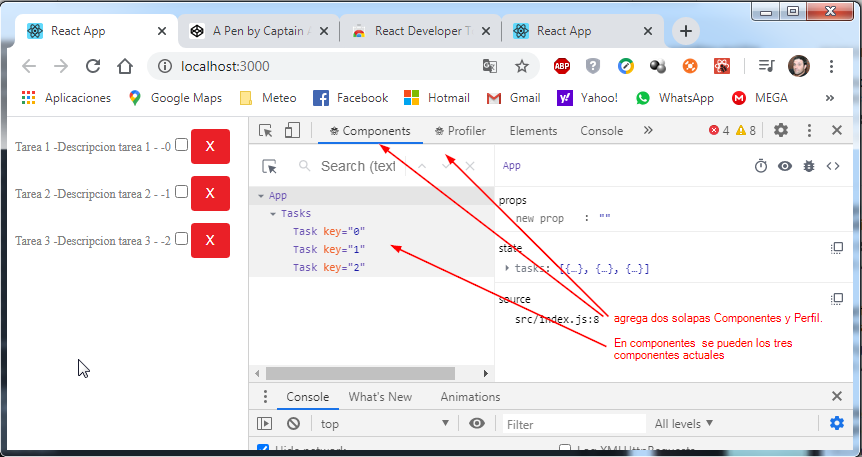


Lo que vamos a hacer ahora es cambiar la propiedad de algún estilo de una etiqueta dependiendo del estado del componente al que esta asociado. Por ejemplo, en una tarea, de acuerdo al estado de ‘realizada’ (true/false) cambiarle el color al texto del párrafo, el fondo, etc.

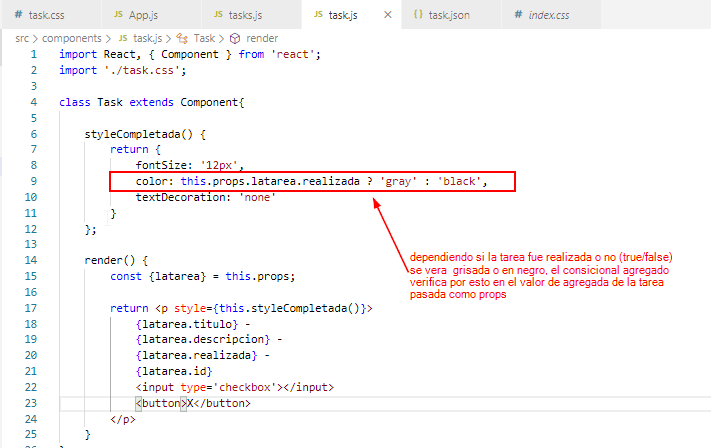
Eso lo vamos a hacer con funciones, es decir en línea.

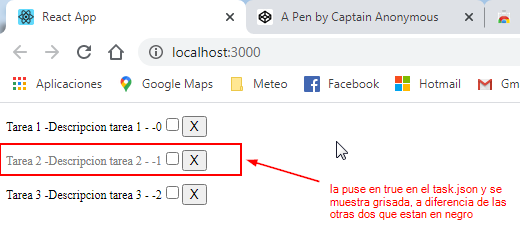
Vamos a mencionar aquí la existencia de un plugin para Chrome de React.js que permite realizar un debbug de código React, se trata de **React Developer Tools**. Este plugin agrega una solapa en las Herramientas del Desarrollador de Chrome que me permite inspeccionar React. Esta solapa aparecerá cuando la página que intentamos analizar esta desarrollada con React. Una de las cosas que tiene es que me permite modificar valores de cosas allí que se ven en la pagina con lo cual puedo debbuguear de manera más rápida mi desarrollo.





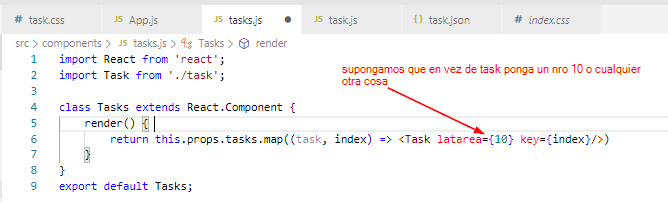
El caso de usar una función en vez de un objeto en un const, es que en una función puedo agregar condicionales, if then, por ejemplo voy a agregar un condicional para que el color del párrafo sea gris o negro dependiendo del valor true o false de completada.



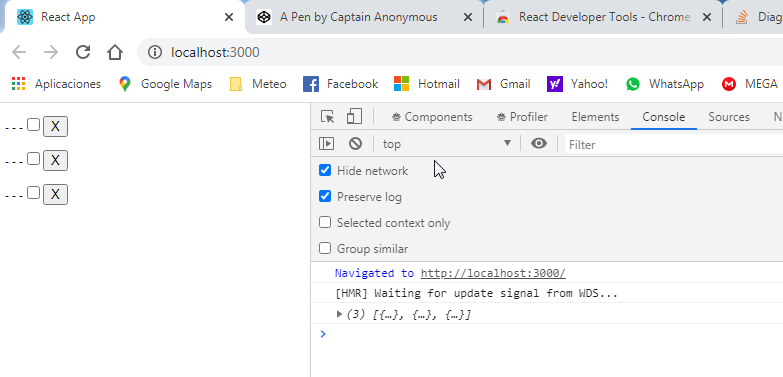


# PropTypes

Hacen referencia a los tipos de datos de las propiedades **props**. Permiten hacer un control d que el tipo de la propiedad que estamos pasando es la que corresponde. A veces uno puede pasar una propiedad que no corresponde y la aplicación no da error pero no funciona como debiera de funcionar.

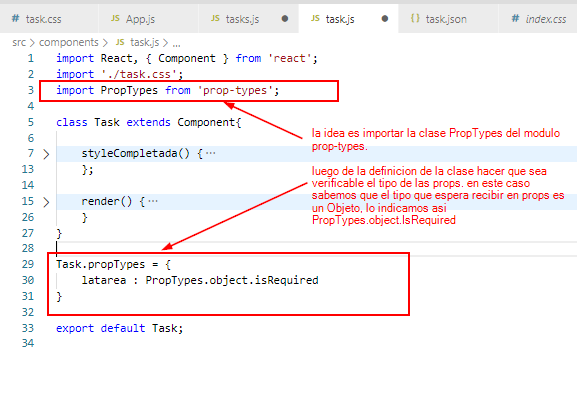


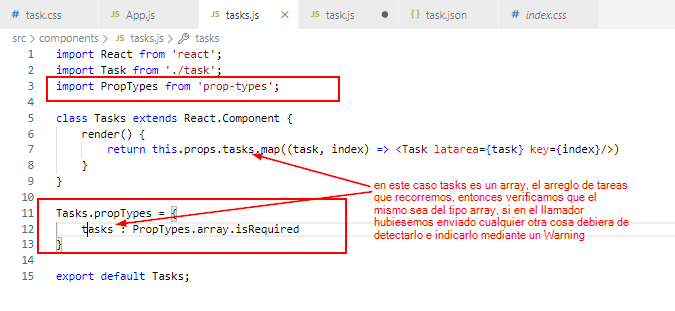
La aplicación funciona, no da error en la consola, pero no muestra lo que debería de mostrar.

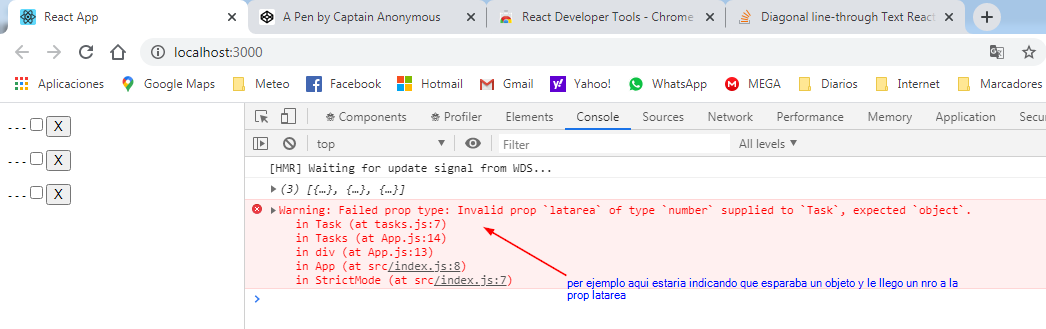


son casos en el que es difícil encontrar donde se produjo el error. Para poder permitir que se haga un control de las props y pueda detectarse un caso de ese tipo es en el código agregando código que me permita indicarle al node que verifique el tipo de las propiedades al momento de compilar si encuentra algo diferente a lo que le indico que debe esperar este lo indicara mediante un Warning por consola.

La idea es importar las clase PropTypes del modulo prop-types y utilizarlo luedo de la definición de la clase para indicarle que oblige a chequear que el tipo de la propiedad sea por ejemplo un objeto, un arreglo, o cualquier otra cosa,







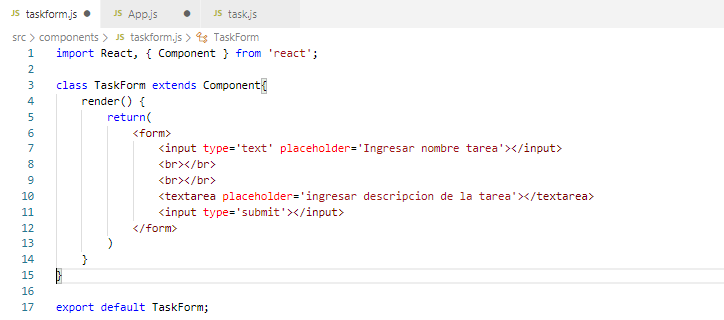
Esto es para tener una idea de que se puede hacer un control de los tipos de las props, pero se puede ahondar más sobre el tema en la documentación de React específicamente en la guía sobre Verificacion de tipos con PropTypes.

# Forms

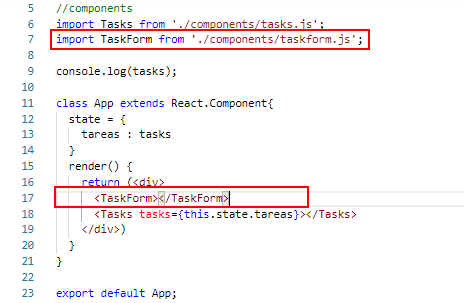
Por lo general cuando trabajamos con una aplicación web es bastante normal que agreguemos formularios o cuadros de texto para que el usuario ingrese información o le permitan hacer algo en la aplicación.

Tenemos en nuestra aplicación una lista de tareas, ahora vamos a agregar un formulario para que de alguna manera le permita al usuario ingresar nuevas tareas. Para esto vamos a agregar un nuevo componente taskform.js en la carpeta /components y le vamos a dar el comportamiento necesario.

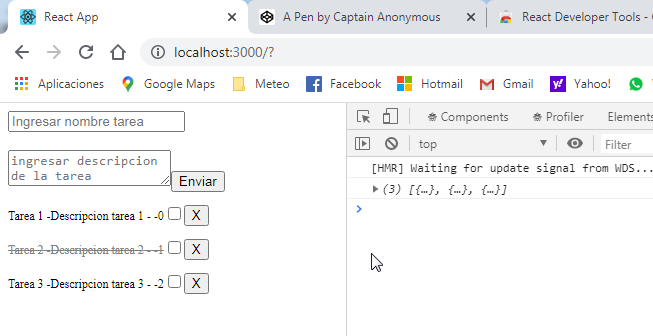
Creamos una clase del componente TaskForm que extiende a Component y en el método render() le agregamos un cuadro de edición <input type=’text’>, un area de texto <textarea> y un botón <input type =‘submit’>. Le damos para importar



En la calse App agregamos este nuevo componente antes del componente de las tareas

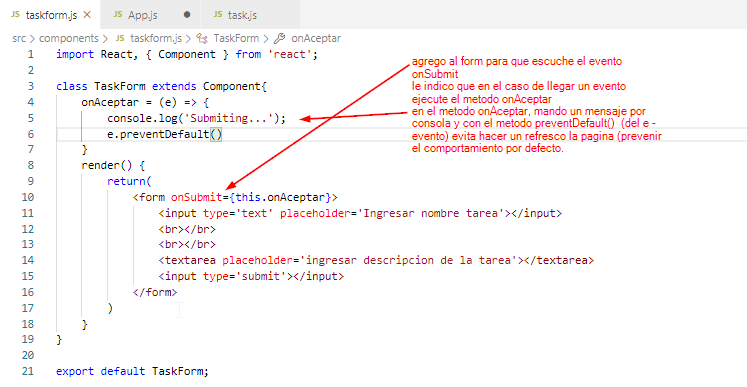


Lo que veríamos ahora sería algo del tipo

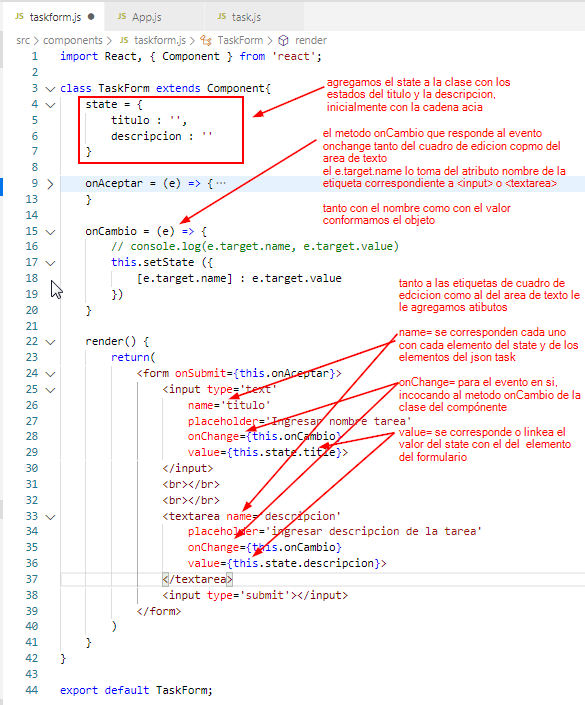


Una de las cosas que primero vamos a ver es que cada vez que se oprime el submit se hace refresca la pagina por completo y eso es una de las cosas que vamos a querer evitar, entonces ataquemos primero el tema de la respuesta al evento del botón submit del formulario. Para responder al evento onSubmit, vamos a escribir un método onAceptar(), en la clase. La forma que tiene menos complicaciones es escribirlo como una función flecha, ya que si lo escribimos como function después hay que utilizar el Bind() para bindear.

Dentro del método, envio un mensaje por consola que vere como efecto de pulsar el voton enviar y además uso en el método el argumento e (event) y de este llamao al método preventDefault() que lo que hace es prevenir el comportamiento por defecto del evento que seria refrescar la pagina, con lo cual se ve que el proceso de recarga de la pagina no se lleve a cabo.



Ahora vamos a escribir un método común para los otros elementos del formulario, de manera que pueda tomar los datos que carga el usuario y dejarlos en el estado (state) del componente.



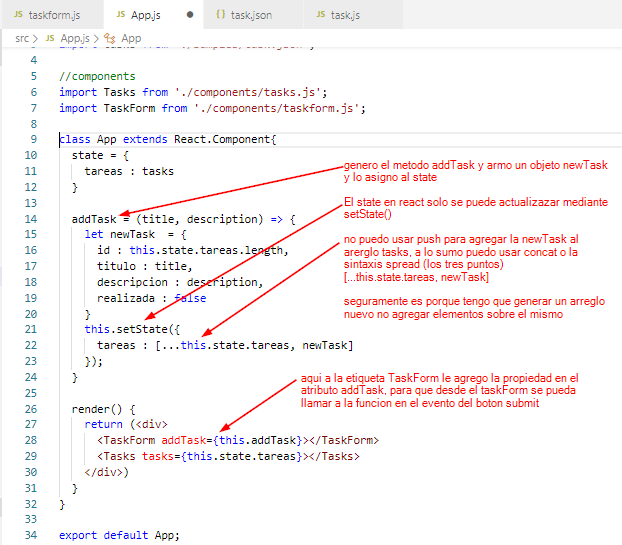
Se agrego un **state** al componente **TaskForm** que guarda relación con los datos que carga el usuario en <imput> y en el <textarea> a estos a su vez se les agrega un atributo **name** que concuerda exactamente con las propiedades del objeto que contiene el **state** y a su vez con las de los objetos que están en el task.json. Además, se agrega a cada uno de ellos el atributo **onChange** para manejar dicho evento, desde este se llama al método **onCambio**() de la clase del componente **TaskForm**. Y finalmente el atributo **value**, que es el que vincula el texto que ingresa el usuario por el teclado (en el formulario) con el **state** del componente.

de Ahora que vemos el objeto generado con los dato de la nueva tarea que ingreso el usuario, estamos listos para ver como agregarlo a la lista tareas.

# Pasar funciones a través de props

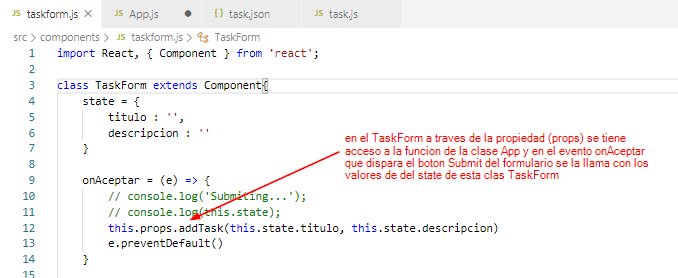
Analizamos un poco, el componente TaskForm es el que tiene la tarea de recibir los datos, veamos entre los demás quen seria el candidato a procesar estos datos, es decir sumarlos a la lista de tareas. EL componente Task, por ejemplo es el encargado de mostrar una tarea determinada, no perece ser este el encargado de guardarla. El componente Tasks pareciera que podría ser, es el encargado de recorrer la lista de tareas, pero no tiene trabaja con los datos, solo los recibe a través de **props** y los pasa. Por último queda App, y en este vemos que es el que posee los datos de las tareas, de todos los que hemos visto es el que posee los datos de las tareas para poder accederlos. En este componente esta el arreglo que tienen las tareas, entonces tenemos que de alguna manera alterar la información, es decir, agregar la nueva tarea al arreglo. En otras palabras lo que obtenga en el formulario voy a tener que pasárselo desde el formulario al **state** de la clase App ya que allí están las tareas. De alguna manera vamos a tener que conectar el TaskForm con el App, y eso será a través de funciones.

Vamos a agregar un método addTask() al componente App y se la vamos a pasar a TaskForm como una propiedad en props, agregando en la etiqueta <TaskForm addTask=’addTask’> es decir, la propiedad se llama addTasck y le manda a travez de ella la funcion addTask()

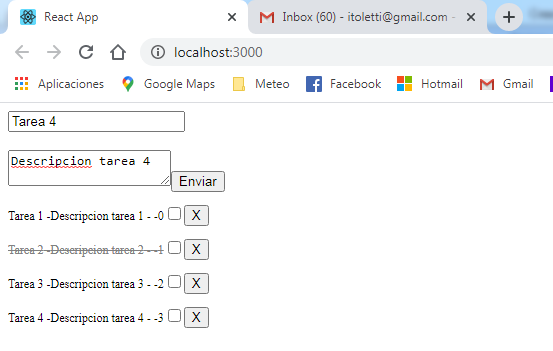


Dentro del método addTask se genera una nueva tarea con los parámetros que se pasan desde el TaskForm y luego esa tarea que es un objeto de la agrega al arreglo de tareas. Lo primero que se me ocurrió es hacerlo con un push(), pero me dio error, buscando documentación del error indicaban hacerlo mediante un concat() a diferencia del push(), el concat() devuelve un Array y quizá es eso lo que necesitamos para el modificar el estado state. Además algo importante es que el state de React solo se puede modificar a través de la función setState(), no se permite hacerlo directamente haciendo mención a this.state.tareas (seria el arreglo de tareas).

Por otro lado desde la clase del formulario TaskForm en el evento onAceptar, lo que hacemos es llamar a la función haciéndole mención a través de props (this.propsaddTask(param , param2). Los parámetros serán los valores del state en el TaskForm.

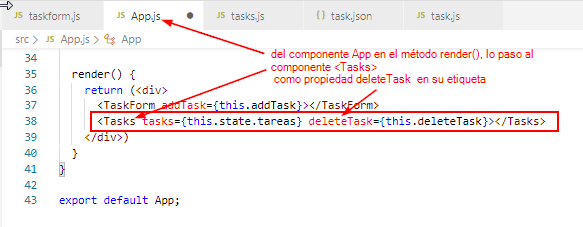


En la imagen de abajo se ve que agregue la tarea 4, que obvianebte se perderá cuando refresque la pagina ya que no la guardo en el archivo task.json, solo se la agregó a la lista de tareas.



Vamos ahora a trabajar para escribir dos métodos nuevos, deleteTask() y checkDone() en la clase del componente App que es donde está el arreglo de las tareas. Y las llamadas a estos métodos hay que ver cuál es el componente que necesita esta función, donde sería factible llamar a estos métodos. El botón de delete lo veo en el componente Task por ende tendré que llamar el método en la clase del componente Task, lo mismo para el checkbox.

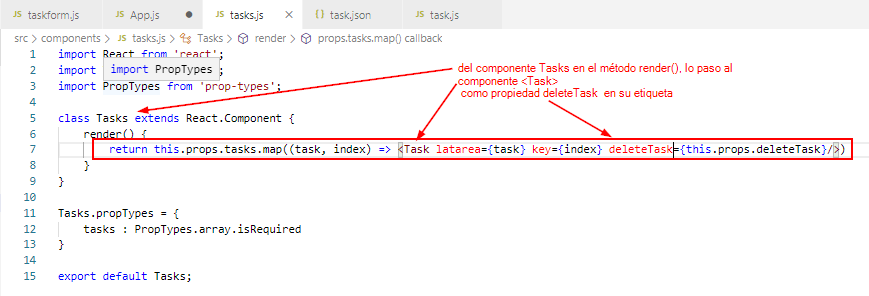
Ahora, veo que el complemento App no tiene conexión con el complemento Task, pero sí se importa el componente Tasks, entonces se le puede hacer llegar a través de éste otro componente. Es decir, le voy a pasar un método da un componente (Tasks) y éste componente se lo va a pasar a otro (Task)

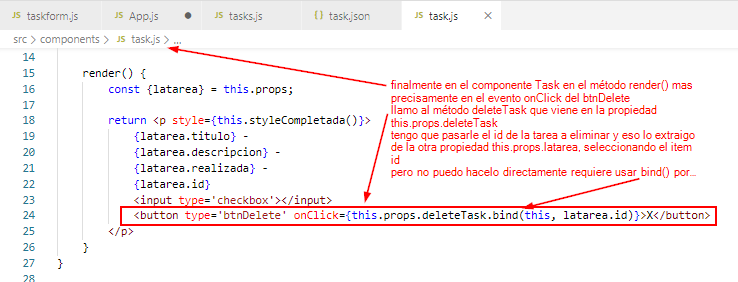


Solo queda agegar una sentencia con el setState() para que el arreglo de las tareas en el estado (state) se vaya actualizando

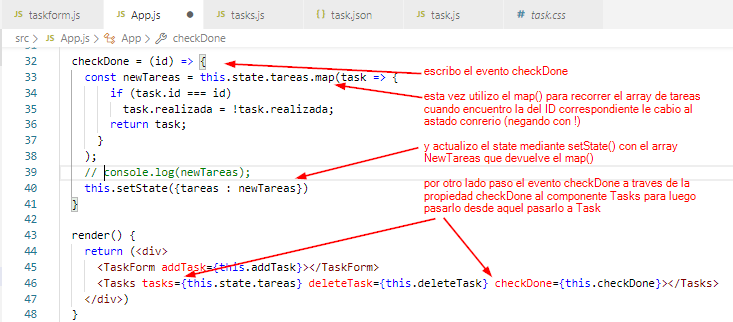
  this.setState({tareas : newTareas})

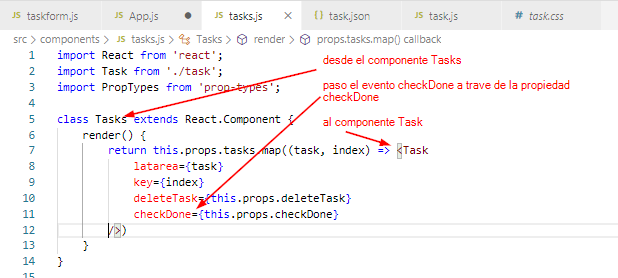
con esto se actualiza el estado y la pagina ira mostrando la lista nueva.

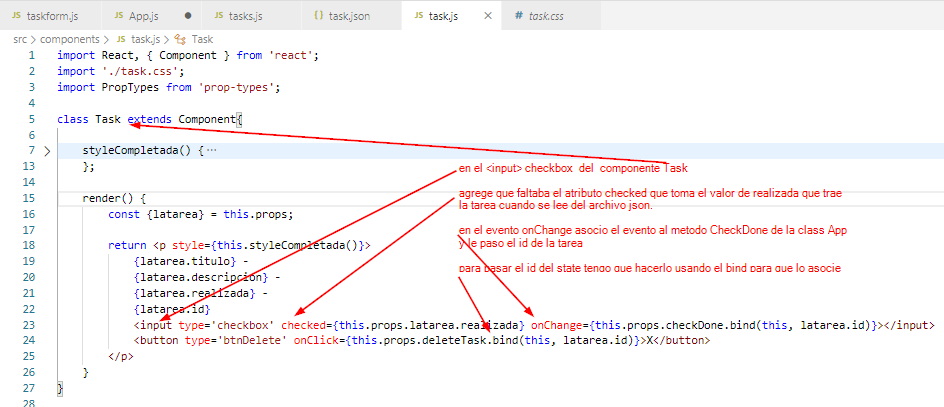




Para los checkbox el tratamiento es similar.







Esto de pasar métodos entre componente es fácil al p´rincipio y cuando la aplicación es chica, pero cuando se trata de aplicaciones grandes las cosas se comienzan a tornar inmanejables. Alli aparece el concepto de Redux, es una biblioteca que permite administrar el estado de la aplicación, en vez de pasar métodos de un lado a otro, va a existior un lugar centralizado donde va a permanecer el estado de la aplicación y todos los componentes van a poder tener acceso a ese estado. Es un concepto que no se va a tratar aquí, pero si se aconseja aprender bien esto de pasar los métodos a través de React y luego si aprender el concepto de Redux porque es bastante utilizado en React, aunque existen otras soluciones aparte de Redux pero es el más popular.

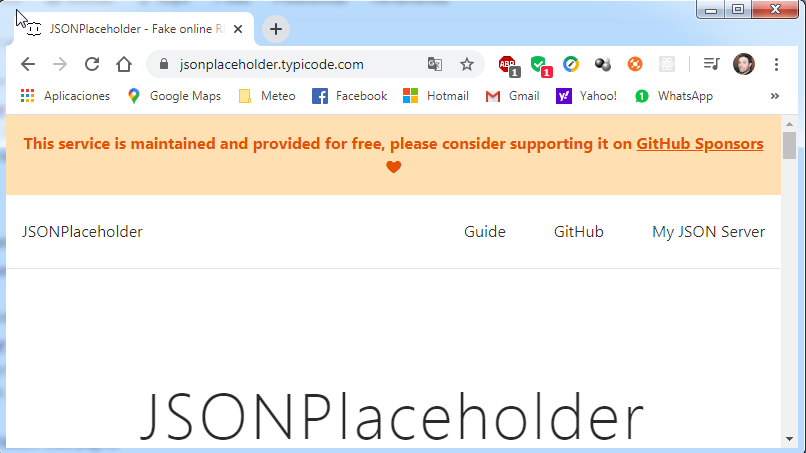
Con esto hemos finalizado esta aplicación inicial.

# Fetching Data

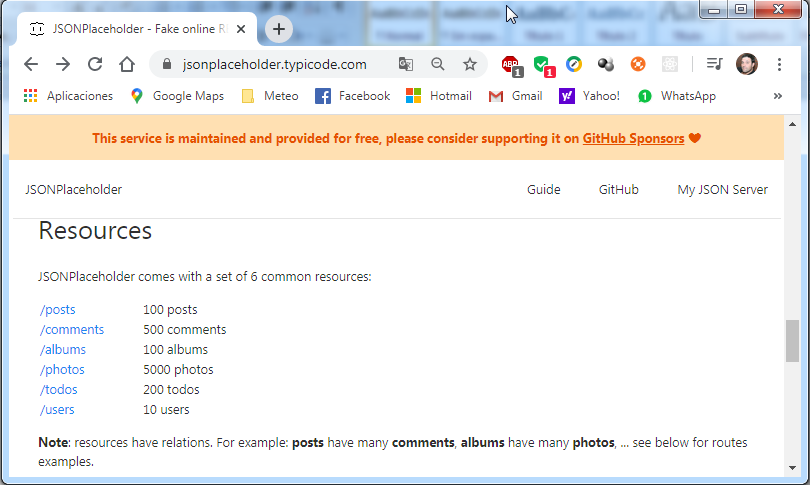
En esta sesión vamos a aprender como recibir datos de servidores externos. Este es el enfoque más común cuando trabajamos con una aplicación de React. Cuando desarrollamos una aplicación que está desarrollada con código de JS, por lo general los datos no residen es esa aplicación. Es decir, cuando trabajamos con un navegador solo estamos trabajando con una interfaz, los datos residen en servidores. Y para programar servidores se utilizan lenguajes de programación de back-end como PHP, Ruby, Python, JavaScrip, Go, etc. En este Caso como estamos trabajando en front-end vamos a necesitar obtener datos de algún back-end ya desarrollado y por suerte existen páginas de internet que nos entregan datos ya desarrollados para que los podamos reutilizar, por ejemplo JSONPlaceholder. JSONPlaceholder es un sitio web en donde podemos obtener datos y utilizarlos en nuestra aplicación.

Por lo general este el enfoque que vamos a utilizar en una aplicación real, un desarrollador back-end va a darte los datos y tu siendo un desarrollador front-end vas a tomar esos datos y los vas a utilizar en tu aplicación de React. Mostrarlos a través de una interfaz, esa es la tarea básica de un front-end.

A través de este sitio web, vamos a poder simular esto, es decir, va a simular nuestro back-end.

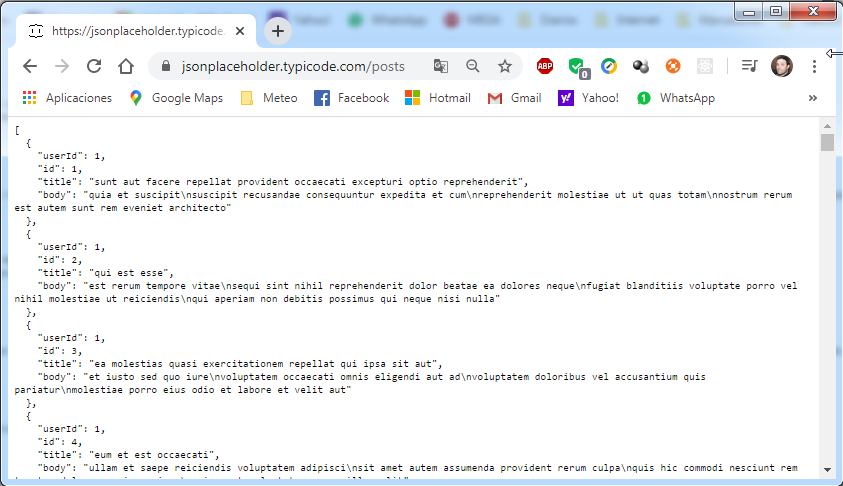


Mas abajo puede verse un listado de los recursos que ofrece y las relaciones que existen entre ellos, por ejemplo 100 posts, donde cada posts posee muchos comentarios, 100 albums que a su vez contienen muchas fotos



Además con el servicio en línea My JSON Server y un simple repositorio de GitHub, puede tener su propio servidor REST en línea simulado en segundos.

Veamos para uno de los recursos, por ejemplo, para /posts tengo 100 datos en formato array de objetos Json en una url: https://jsonplaceholder.typicode.com/posts



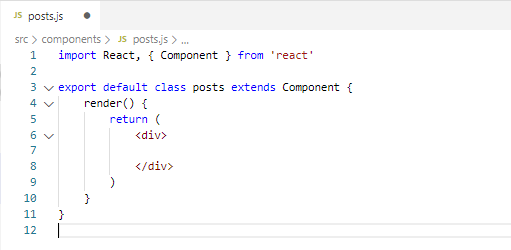
¿Como podemos hacernos de estos datos?

Simplemente haciendo una petición http a la URL del recurso (que quiera utilizar). Usando una API fetch() del navegador voy a poder hacer una petición http a dirección (URL) y cuando la haga me va a devolver todos los datos que se ven allí.

Vamos a verlo en nuestra aplicación, pero vamos a crear un componente nuevo, podríamos hacerlo en el que ya hemos creado pero puede crear confusiones, ya que los datos de posts si bien son parecidos a los de tareas no son lo mismo.

Creamos un componente nuevo posts.js en la carpeta /componentes e importamos React…

pero vamos a hacerlo de una manera particular. Como viemos en los antyeriores componetes hemos tenido que importar React y component crear la clase y escribir el método render. Es un trabajo repetitivo, pero evitarlo podemos adicionar un plugin (extensión) a VSCode el **VS Code ES7 React/Redux/React-Native/JS snippets**, que de acuerdo a abreviaciones escribe ‘retazos’ de código repetitivos por ejemplo con escribir rcc dentro del archivo posts.js, ya nos autocompleto el código necesario para crear un componente post. Hay muchas otras abeviaciones para crear cosa habitualmente utilizadas con Ract y para conocerlas hay que ir a la documentación de esta Extension



Ahora si vamos a lo que queríamos hacer desde un principio. Ya tenemos el nuevo componente agregado ahora lo que queremos es mostrar una lista de los posts que nos da la url de posts de JSONPlaceholder. Entonces, antes de reendderizar algo, lo primero que tendríamos que hacer es obtener esa información. Esos datos los quisiéramos obtener antes de que cargue la aplicación para que, una vez obtenidos los datos, ya pueda ‘pintarlos’ en la página.

Nota aparte - Existe algo llamado lifecycle method en React

¿**Cuáles son los métodos de React lifecycle**?

Puede pensar en los métodos del ciclo de vida de React como la serie de eventos que ocurren desde el nacimiento de un componente React hasta su muerte.

Cada componente en React pasa por un ciclo de vida de eventos. Pensar que pasan por un ciclo de nacimiento, crecimiento y muerte.

**Montaje**: nacimiento de su componente

**Actualización**: crecimiento de su componente

**Desmontar**: muerte de tu componente

* El render() es el método de ciclo de vida más utilizado.

Es una función pura.

No puede establecer el estado en render ()

* ComponentDidMount() ocurre tan pronto como se monta su componente.

Puede establecer el estado aquí pero con precaución.

Este es un buen lugar para iniciar llamadas API, si necesita cargar datos desde un punto final remoto.

* ComponentDidUpdate() ocurre tan pronto como ocurre la actualización.

Puede establecer el estado aquí pero con precaución.

* ComponentWillUnmount() ocurre justo antes de que el componente se desmonte y se destruya.

Este es un buen lugar para limpiar todos los datos.

No puede establecer el estado aquí.

* ShouldComponentUpdate() puede usarse raramente.

Se puede invocar si necesita decirle a React que no vuelva a renderizar para un determinado estado o cambio de accesorio.

Esto debe usarse con precaución solo para ciertas optimizaciones de rendimiento.

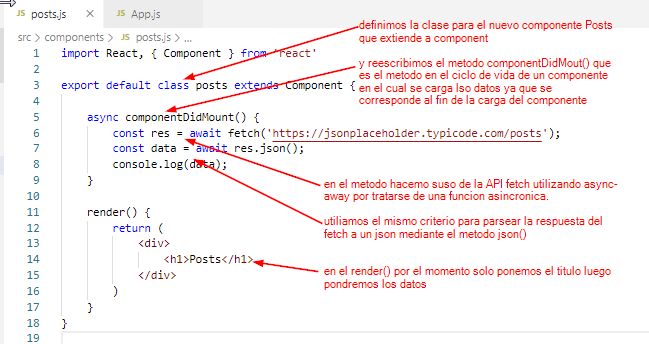
Los dos nuevos métodos de ciclo de vida son getDerivedStateFromProps () y getSnapshotBeforeUpdate ().

Deben usarse solo ocasionalmente.

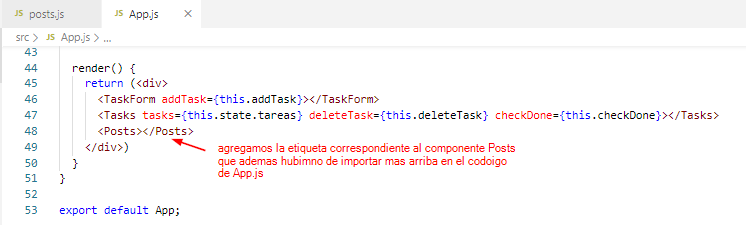
No hay muchos ejemplos disponibles para estos dos métodos y todavía se están discutiendo y tendrán más referencias en el futuro.

## Fin nota aparte

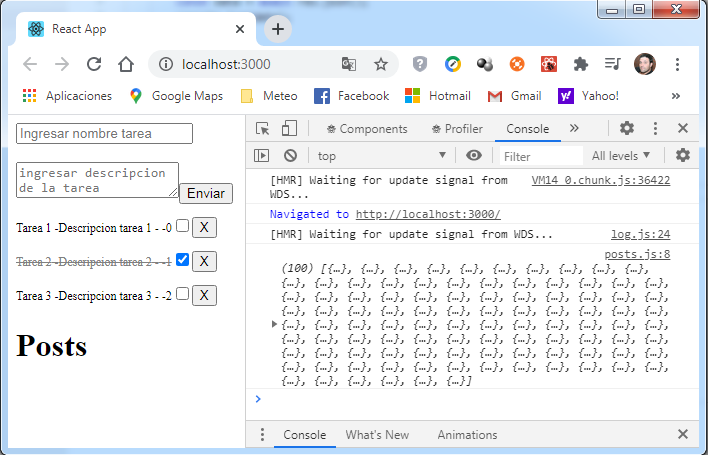
Entonces, según lo anterior la captura de los datos desde el back-end ficticio va en el metodo del ciclo de vida del componente React componentDidMount(). Escribimos ese método en la clase del nuevo componente Post y en el hacemos el fetch() a la URL que nos indica el JSONPlaceholder para los posteos. Como ya sabemos el fetch() demora por lo cual es asíncrona, por lo cual utilizamos async-away. Tambien sabemos que la respuesta del fetch biene en un formato que debemos transformar a Json por lo cual debemos aplicarle el método json() para parsearla. Quedaria algo así



Por otro lado en App.js tenemos que sumar este componente al los dos que ya teníamos el (formulario y la lista de tareas)



hasta ahí nomas, solo imprimiendo por pantalla el resltado del fetch() luego de haberlo pasado a json, veo en la consola del navegador el arreglo con 100 elementos.

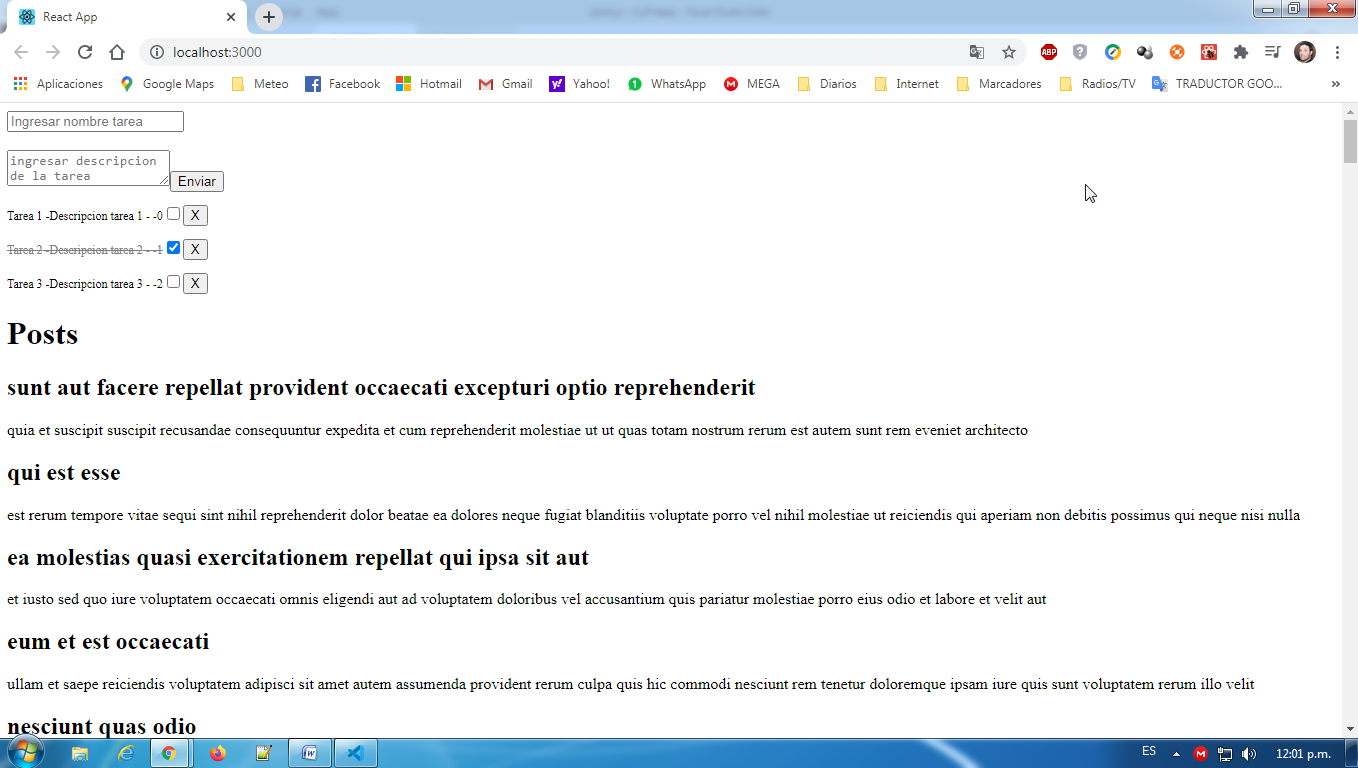


Vamos a crear un **state** posts en el componente, inicialmente será un array vacio y allí vamos a dejar el resultado del fech() mediante el uso de setState().

Luego para utilizar los datos almacenados en el state para ‘pintarlos’ es algo tan sencillo como leerlos en un bucle for en el método render(). En realidad utilizo un map() para recorrer cada post (envio).



ahora en el componente Posts comienzan a verse los datos del titulo y el cuerpo de los envíos a demás a cada envio que se encapsula en un bloque <div> y a esta etiqueta se le suma un atributo key con le id del envio de manera de poderlos distinguir. Hay que tener en cuenta que ya estos datos pertenecen a esta aplicación sino que provienen de un servidor externo desde JSONPlaceholder. De esta manera es como vamos a trabajar cuando se obtengan datos desde un back-end, por lo general esa parte la desarrolla un programador back-end



Que hay entones ahora de guardar datos. Para guardar datos se utilizan los métodos http POST, para eliminar DELETE, o para actualizar PUT. Pero estos no son conceptos que están relacionados netamente con React, si bien son importantes en React, están más bien relacionados con JS ya que en Riact no hacemos más que utilizar código JS, como hemos visto.

Ahora en la última sección vamos a ver cómo utilizar el enrutador de react.

# React Router

Hasta este momento, tratando de aprender el uso de React, desarrollamos una parte usando un aplicación de tareas y en otra usando una lista de posteos (o publicaciones), es decir hemos desarrollado dos funcionalidades que no tienen mucho que ver. Sería bastante bueno si pudiésemos dividir esas funcionalidades en ‘múltiples paginas’… en ese caso no podemos dividirla en múltiples páginas pero si podemos simularlas, es decir podemos mostrar un contenido u otro a través de una navegación. Para esto vamos a tener que conocer un concepto de React llamado enrutador, que es bastante sencillo.

Vamos a necesitar importar cosas de la librería de React para desarrollo web react-router-dom (existe otra para desarrollo móvil) en App.js. De esta librería en particular vamos a necesitar tres cosas

1. el BrowserRouter es el enrutador, como el nombre es largo lo renombramos como Router
2. Route, para definir como una etiqueta las diferentes rutas
3. Link, para crear una especie de navegador

Estas modificaciones van dentro del método render() de la clase App, cuando se pinta la pagina.

En principio creamos un bloque con etiqueta <Router> donde encapsulamos el menú de navegación <nav> y las dos rutas en bloques <Route> que nos permitirán acceder:

1. <Route Path=’/’ > que rutea a la raíz que sería la pagina de las tareas donde están, el componente del formulario, de etiqueta <TaskForm> y el componente de la lista de tareas , de etiqueta <Tasks>.
2. <Route path=’/posts’> que rutea a posts, que sería la pagina de los posteos, el componente de la lista de posteos, de etiqueta <Posts>

Para la primera ruta se escribió con un (render() para pintar el Form y la lista, en camino en el segundo se hizo directamente indicando el componente en el atributo Component del la etiqueta Route.

Seguramente se pueda hacer con el render() Como el primero pero al ser uno seguramente no tiene sentido, habría que ver si el primero se puede de alguna manera hacer de esta forma también.

