Curso de React

¿Qué es React? A JavaScript library for building user interfaces Es Una librarería de JavaScript que nos va ayudar a construir interfaces de usuario.

Las necesidades del Frontend han ido en aumento a medida que han pasado los años. Todo el enfoque ha sido encaminado a mejorar todos los aspectos que hagan que la experiencia de usuario sea mucho mejor, esto le ha dado más responsabilidades al área de Frontend.

Siempre hemos manejado el dinamismo conn javascript, pero si vamos a manejar el dinámismo y la interfaz al mismo tiempo entonces javascript se va a comer la interfaz, entonces tendría que redefinir un poco de como hacemos esto. Para interfaces super dinámicas esta es una solución interesante.

Historia: jquery era muy famoso en el año 2005, si hoy por hoy sabemos que javascript no se interpreta igual en todos los navegadores, por eso jquery era la solución ya que su filosofia era la de **'escribe menos, haz más'** una librería que se encarga para hacer ese código diferente para cada navegador y hacer que tu solo escribas una sola función que brindaba jquery y te evitabas ese problema de la validación por cada navegador, esto era super interesante y esto era un super poder que se apreciaba muchisímo en esa epoca.

Las interfaces empezaron a tener más datos, empezaron a tener más responsabilidades porque otra vez jquery nos ayudaba a hacer imagenes, empezamos a construir un poco de interfaces con jquery con su método de html, para eso llegabaman más apoyos a jquery que ya nos ayudaba a crear la interfaces, para esto llega 'backbone.js' hallá en el 2010 cuando fue muy popular. Backbone nos ayudaba a interactuar con datos a manejar como por ejemplo un lista. Y para el dinámismo jquery era el motor de esta librería.

Como el frontend comienza a tener más y más responsabilidades y también los frontends más responsables con la interfaz de usuario y cuidarla y hacer que la aplicación pesará muchisímo menos e hiciera mucho más, para esto en vez de combinar librerias y hacer que una librería sea dependiente de otra, empezamos a crear estos frameworks.

En el 2012 se presentan la solución de {Angular, ember y meteur} y muchas otras más. Se crean estos frameworks para ayudarnos en lo que hacia backbone y jquery juntos pero sin usarlos como dependencias amigas sino hacernos una propia interfaz para resolver estos problemas es decir Angular, Ember y meteur tenían su forma de resolver los cliks del usuario, la forma de enviar datos, es decir todas esas herramientas que necesitas para crear interfaces con usuario, ya las hacian ellos todo en uno.

Pero cada día se van optimizando como se hacen las aplicaciones dentro del cliente y ahí es donde entran nuevos jugadores a la mesa: Aquí entra el protagonista de esté curso React, Vue.js Angular. Otra vez angular pero en su versión 2, aunque sigue siendo angular hubo un cambió muy drastico entre la versión 1 y 2, al punto de que hay personas que mantienen aplicaciones completas en angular 1, y gente que está empezando proyectos en agular 2 o ya mantien proyectos ahi porque son totalmente diferentes.

Esté enfoque el actual es la última generación de librería y frameworks de javascript para construir interfaces es muy interesante porque tiene un enfoque orientado hacia componentes, es decir todas la vida hemos dividido en css como: header, footer, navegación, elementos hijos, elementos padres, pero llevar esa lógica a como se contruían las interfaces y como se dividían para dividir responsabilidades. Es un enfoque muy interesante para que una aplicación pueda ser más mantenible y tanto Agular, React y Viue las tienen.

- Particularidades de React.js

- Declarativo: Es muy sencillo de escribir interfaces y poderlas leer.
- Basado en componentes: crear componentes es como jugar con legos.
- Aprende de una vez y escribe donde quieras.

Con React.js buscamos que nuestra jenga no se caíga, sino que permanezca homogenea y que la podamos construir, que podamos sacar, que podamos sacar fichas y que esto no se caega, que todo se mantenga separado y junto al mismo tiempo. Separado de que cada cosa pueda vivir por su cuenta y junto, porque en conjunto harán un aplicación gigante que será la que amarán tus usuarios y esto es algo que hace react.

Otra cosa que vamos a ver con react.js es que como esto es una librerpia solo se encarga de 1 sola cosa, y es de como construir interfaces. Construir interfaces y hacerlas un poco dinámicas a nivel de interacción con el usuario, **pero hay otras librerías que pueden complementar a react.js** como: next.js, reactRouter, redux, react-native.

Requisitos previos para el curso

React es una librería de javascript y necesitamos una forma de instalar esa librerpia de javascript y esa forma es con npm. Npm es la forma de instalar cualquier librería o cualquier framework dentro de javascript, ya se volvió standart en la industria, pero npm viene incluido dentro de node.js

Así que vamos a instalar node.js que a su vez nos va a instalar npm y con esto ya tenemos lo básico para empezar con el curso.

Solo ingresa a la página de node.js y solo descargalo e instalalo siguiendo las instrucciones de la página.

Hay 2 formas de hacer aplicaciones en react, bueno 2 formas sensatas de hacer aplicaciones en react.

1. Es hacer un bundlerplay acerca de como construir una aplicación dentro de react que la puedes encontrar en create-react-app. Esto lo hizo la comunidad en conjunto para que iniciar con un proyecto en react y un proyecto así de ambisioso no sea tan complicado, simplemente se instala y asi lo tendremos en nuestro sistema instalación: `npm install -g create-react-app`

Ahora con create-react-app podemos empezar nuestra aplicación simplemente irnos a la carpeta dode queremos iniciar nuestra aplicación usando el comando 'create-react-app' seguido del nombre de nuestra app `create-react-app my-app`

Luego entramos a nuestra aplicación con cd my-app Despues creamos nuestro packages.json haciendo un `npm init` dentro de nuestra aplicación.

2. Otra forma de configurar react.js es que tú hagas tu propio setup desde cero, configurando tu webpack y tu packages.json, instalando tus dependencias y haciendo algo custom para ti, esté es el camino correcto para aplicaciones de producción, pero para hacer apliaciones rápidas y probar nuevas ideas seguramente create-react-app es la mejor.

Configuración de Webpack para React

Ya sabemos como arrancar con un proyecto de webpack de una forma sencilla gracias a create-react-app pero que tal si tu quieres hacerlo de una manera custom, es decir si queremos añadir nuestra propia personalización acerca de como obtener el setup, la configuración inicial de nuestro proyecto y dominarlo al

completo o simplemente quieres empezar un proyecto super ambicioso y quieres obtener todo el control, para esto haremos la configuración con webpack.

si quieres aprender más acerca de webpack puedes entrar el curso de webpacl

En esté curso hicimos una configuración perfecta para webpack que pusimos en uno de nuestro proyecto finales que fue pasar invie, un proyecto en el curso de animaciones para la web a webpack, está configuración la puedes encontrar en mi github donde documente todo el contenido del curso y las disitintas configuraciones que pudes tener solo ingresa a webpack-course

En la carpeta de invie vamos a copiar 2 archivos para nuestro proyecto de react y son 'webpack.config' y webpack.dev.config.js porque vamos a tener 2 entornos 1 para desarrollo y 1 para producción, uno para que el código corra super rápido y podamos depurar nuestro entorno y otro de producción donde el código corra super rápido pero en el navegador y que lo entiendan todos los navegadores, solo vamos a descargarlos y empezar a configurar nuestro entorno

Otra cosa que nos falta es que estas configuraciones requieren de otros modulos de npm que son solo utilizados en un entorno de desarrollo

¿Qué quiere decir que solo sean utilizados en un entorno de desarrollo?

Esto quiere decir que nos van a ayudar a compilar nuestro código bonito que vamos a escribir y solo nos van a ayudar para generar un build, un archivo que vamos a lanzar despues al navegador es decir estás dependencias solo tienen que vivir en el lugar donde estan compilando estas, en esté caso mi computador o en tu caso puede ser el servidor. Para eso debemos iniciar nuestro proyecto en un inicio y para esto no va a ayudar npm

Solo tenemos que ir a la terminal y ejecutar el comando `npm init`

Nuestros archivos de webpack.config necesitan algunos modulos extras para funcionar, para ello tenemos que volvernos a nuestro proyecto que hicimos con webpack y copiar las dependencias de desarrollo que necesita nuestros archivos de configuración de webpack para poder compilar nuestros archivos.

```
"devDependencies": {
    "@babel/core": "7.0.0",
    "@babel/plugin-proposal-object-rest-spread": "7.4.4",
    "@babel/plugin-proposal-pipeline-operator": "7.3.2",
    "@babel/plugin-transform-spread": "7.2.2",
    "@babel/polyfill": "7.0.0",
    "@babel/preset-env": "7.4.5",
    "@babel/preset-react": "7.0.0",
    "babel-loader": "8.0.6",
    "babel-plugin-transform-object-rest-spread": "6.26.0",
    "clean-webpack-plugin": "0.1.17",
    "css-loader": "2.1.1",
    "extract-text-webpack-plugin": "4.0.0-beta.0",
    "file-loader": "3.0.1",
    "url-loader": "1.1.2",
    "webpack": "4.32.2",
    "webpack-cli": "3.3.2",
```

```
    "webpack-dev-server": "3.5.1"
  }
```

Lo más recomendable es fijar las dependencias que vallamos a ocupar en nuestro proyecto para que en el futuro nuestro proyecto actualice las dependencias ya que las nuevas actualizaciones pueden romper nuestro código ya que las implementaciones cambian. Solo algunas veces puede fallar pero lo recomendables es no actualizar las depencias por si solas.

Ahora que tenemos estas dependencias declaradas en nuestro archivo package.json porque no las instalamos porque todavia no las hemos instalado dentro de nuestro entorno dentro de nuestra carpeta, simplemente vamos a ir a nuestra terminal y vamos a instalar nuestra dependencias con el comando `npm install`

mientras se instalan vamos a hacer un par de configuraciones adicionales, algo que requiere nuestro webpack.config y algo primordial de el, es cual va a ser mi proyecto, y donde está úbicado porque que arhivo voy a compilar porque todavía no hemos escrito absolutamente nada. Y este archivo en el curso anterior era invie, pero ahora tenemos uno diferente que se llamá platzi-video y que tal si modificamos nuestro entrypoint

```
entry: {
    'platzi-video': ['babel-polyfill', path.resolve(__dirname, 'index.js')],
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'js/[name].js'
  },
```

También tenemos que copiar los scripts que ya hicimos en el curso de webpack, en este curso no veremos webpack a profundidad, para eso ya existe el curso de webpck, aquí solo nos centraremos en aprender react al 100%

Usando ReactDOM y JSX

Nuestro proyecto de platzivideo en el entorno en su setup en su configuración previa ya esta listo, pero ahora queremos utilizar react sobre esté porque si bien acá en el navegador podemos soportar código de javascript moderno, lo que todavía tenemos que hacer para soportar react es instalar react y las dependencias que sean inherentes de react para poder utilizarlas, para eso vamos a ir a nuestra terminal y vamos a instalarlas con npm. instalación: `npm install react react-dom --save` Ya que tenemos esto listo ya podemos empezar a trabajar en nuestro proyecto pero con react y react-dom, ¿Porque son 2 dependencias y no solo 1? porque cada quien se va a encargar de una cosa en especial y eso lo veremos más adelante con forme vallamos escribiendo código .

Una vez instalado vamos a empezar a usar react, para ello vamos a ir a nuestro index.js.

- Primero necesitamos importar react
- También vamos a importar react-dom

```
import React from 'react';
import ReactDOM from 'react-dom';
```

REACT: Me va a servir para crear los componentes es decir esos pedacitos de la aplicación para dividir nuestros bloques, nuestros legos. ReactDOM: Me va a servir para poner esos legos en algun lugar para renderizarlos, en esté caso lo va a poner dentro del navegador.

ReactDOM tiene un método que se llamá **render** esté render recibe 2 párametros.

```
ReactDOM.render(¿que voy a renderizar?, ¿donde lo haré?)
```

Son 2 preguntas que tenemos que responderle a reactDOM.render, Lo que vamos a **renderizar** puede ser un **componente de react** o un **elemento de react**, donde lo renderizará tiene que ser un lugar en el DOM que ya exista. Tenemos que pasarle ese elemento a javascript y la manerá más adecuada para hacerlo es por medio de su id, aunque también lo podemos hacer por su clase.

```
const app document.getElementById('app');
const holaMundo = <h1>Hola mundo!!</h1>
ReactDom.render(holaMundo, app);
```

Como estamos ocupando la configuración que creamos en nuestro curso de webpack, tenemos 2 configuraciónes 1 para desarrollo y una para producción, en este momento estaremos ocupando la de desarrollo ya que usuamos un servidor que actualiza nuestra página sin recargar el navegador y eso es muy práctico cuando estamos desarrollando.

Creando Componentes en React

Primeramente vamos a empezar a modular nuestra aplicación para tener todo bien organizado y de está manera tener un código más mantenible y legible, y para construir nuestra aplicaciones solo importaremos nuestros modulos.

Podemos crear componentes en 3 modos:

- Componente funcional
- Componente Puro
- Componente normal o clasico.

Empezamos creando un componente clasico, que es una clase que extiende de react.Component, es decir la clase en esté caso Media va a obtener los poderes de react.Component. Está clase base tiene un método principal que se llamá cuando este sea instanceado y esté método y esté método render va a tener todo esté código que va a ser comoel html que va a tener nuestro componente, es decir su forma su figura, su IA, para eso nuestro render va a retornar el html, es decir podemos poner el html dentro de un return para que sea más legible.

Algo adicional que estamos haciendo es creando esté componente dentro de una carpeta que se llamá src/playlist/component/media.js, esté archivo media.js tiene que ser llamado en el archivo index.js, así como hemos llamado a react como react-dom tenemos que llamar tanto a media. Y la forma en que podemos importar media es que yo pueda exportar algo de esté archivo.js porque no se autoejecuta simplemente tenemos acá un código en este caso tenemos un componente que vamos a exportar, que vamos a enviar cuando este componente lo importe.

Este componente se exporta gracias a Ecmascript 6 gracias 'export default Media;'

```
import React from 'react';

class Media extends React.Component {
  render() {
    return(
      <div>
        <div>
          <img
            src=""
            alt=""
            width={260}
            height={160}
          />
          <h3>¿Por qué aprender React?</h3>
          <p>JasanHdz</p>
        </div>
      </div>
    )
  }
}

export default Media;
```

Ahora si lo podemos importar en nuestro index.js y hacer que react-dom renderé nuestro componente Media que acabamos de crear. Algo adicional es que Media no funcionar tal cual lo estamos recibiendo en el import, ya que un componente para que renderice con ReactDom, debemos renderizarlo como si fuera un tag html o algo parecido el cual Media se cierra en sí mismo. De está manera react sabe que eso es un componente y lo va a renderizar.

```
import React from 'react';
import ReactDOM from 'react-dom';
import Media from './src/playlist/components/Media.jsx';

// ReactDOM.render(que voy a renderizar, donde lo haré)
const app = document.getElementById('app');
ReactDOM.render(<Media />, app);
```

Algo Adicional que podemos hacer dentro de nuestro componente Media gracias a una habilidad de ecmascript6 que es la parte de descomponer un objeto es traerme el Component y no 'React.Component' eso

nos va ayudar a tener un código mucho más resumido y mucho más legible que es lo que siempre buscamos los desarrolladores. Asi que nosotros podemos importar aparte de react, importar {component} dentro de llaves.

```
import React, { Component } from 'react';

class Media extends Component {
  render() {
    return()
  }
}
```

Aunque lo recomendable cuando apenas estamos aprendiendo react es hacerlo de la manera más larga, para saber de donde viene Component.

Estilos con React

Ya creamos nuestro primer componente pero esté se ve con muy poco impacto, lo que vamos a hacer ahora es darle forma a esté componente, de ponerle estilos. Para poner estilos en html hay 2 formas, una es creando una hoja de estilos, una es darle forma a todos lo elementos o puedes ponerle los estilos en linea, los estilos en linea funcionan como un atributo, un atributo que se llama style el cual recibe dentro de comillas los valores.

Algo similar podemos hacer en react, los estilos in-line y también podemos utilizar obviamente css.

Los estilos in-line dentro de nuestro componente van en un atributo style pero en lugar de llevar comillas llevan llaves, que es donde vamos a llamar a nuestra variable, la cual es un key de un objeto de variables para nuestro componente, el cual es declarado antes de returnar nuestra IA o componente. ejemplo.

```
import React from 'react';

class Media extends React.Component {
  render() {
    const styles = {
      container: {
        fontSize: 18,
        backgroundColor: 'lightgray',
        cursor: 'pointer',
        width: 260,
        border: '2px solid red',
        // padding: 12
      }
    }
    return(
      <div style={styles.container}>
        <div>
          <img
            src="./images/covers/bitcoin.jpg"
            alt=""
```

```
            width={260}
            height={160}
          />
          <h3>¿Por qué aprender React?</h3>
          <p>JasanHdz</p>
        </div>
      </div>
    )
  }
}

export default Media;
```

En Javscript vamos a crear un objeto con variables las cuales serán a su vez un objeto que contendran los estilos de nuestra variable, si recordamos en css tenemos estilos conformados por más de una palabra y si nosotros colocamos los estilos usando la conveción de css nuestro objeto js se va a romper, lo que si podemos hacer es encerrar el string que contiene más de 2 palabras en entre comillas o podemos usar la convención que usa javascript que es usar CamelCase.

Camel Lower Case: es si nuestra variable se compone de más de 1 palabra, la primer palabra sea en minuscula y la segunda palabra debería empezar con mayúscula y así sucesivamente ejemplo: backgroundColor: blue o simplemente podemos encerrar los valores entre comillas, y esto es más común cuando le damos valores númericos combinados con nuestra variable de medida. Si la únidad que vamos a ocupar son **pixeles** podemos simplemente escribir el número ya que js lo interpretará la únidad de medida como pixeles, en caso contrario si debería llevar comillas.

Aunque podemos tomarnos toda la vida poniendole estilos en linea, así que hay otra manerá de ponerle estilos a nuestro componente y es a traves de los clásicos .css y también los podemos usar en nuestro archivo de react, gracias a la configuración de nuestro webpack que ya tenemos lista es que ya podemo importar archivos css dentro de nuestros archivos javascript y en nuestro archivo media.js no es la exepción, simplemente podemos importar un archivo media.css y agregarle sus estilos.

Podemos ver que en cada componente va a ir teniendo su propio archivo css junto a el que lo va a cuidar, lo va a respaldar y conservar sus estilos.

Los archivos externos de css hacen referencia a clases dentro de nuestro componente, en jsx tenemos diferencias de como es el html normal ya que **class** es una palabra reservada del lenguaje la cual hemos utilizado en la parte de arriba para crear nuestro componente, entonces nosotros no podemos poner clases de esta manera dentro de los componentes. Para hacerlo simplemente tenemos que **remplazarlo** por **className**

Podemos aplicar más cambios que hay dentro de nuestro media pero simplemente sería aprender más de css pero no más de como funcionan nuestros componentes, así que en este proyecto ya tenemos los estilos listos para que nos centremos en lo que venimos a aprender en Javascript y React.js, no nos detrendremos mucho en los estilos pero nuestra aplicación no se va a ver mal.

Propiedades en ReactJS

Hasta esté momento le hemos puesto algunos atributos a nuestros elementos de JSX de react para darles un poco de estilos. Ahora enfasis en la palabra atributo, en react a los atributos de html no se les llama atributos, sino propiedades, es decir que son las propiedades que tinen nuestro componente y las propiedades que tiene cada uno de los elementos. Está propiedades se las podemos pasar a nuestro componente para decirle que contenido es dinámico y que esto nos sirva como un cascaron de nuestro componente que va a recibir múltiples valores para poder así poner multiples componentes de media que es lo que queremos construir dentro de nuestra aplicación, teniendo eso claro que tal si empezamos a pasarle propiedades dinámicas a nuestro componente de media.

¿donde se renderiza media? dentro de nuestro index.js que simplemente se esta renderizando sin más, a esté media se le pueden pasar propiedades como ahora las conociamos como atributos y aunque son atributos html, en jsx son propiedades siempre.

En nuestro componente podemos pasarle propiedades dentro de la misma etiqueta que envuelve el componente media, ejemplo:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Media from './src/playlist/components/Media.jsx';
const app = document.getElementById('app');
ReactDOM.render(<Media title="¿Qué es responsive Design?"/>, app);
```

Está propiedad la vamos a consumir dentro media, porque ya se la acabamos de enviar, pero este valor viene como propiedades. Otra cosa que tenemos dentro del elemento media es que es una clase. Así que esa clase va a venir provista de algo que se llamá **'this'** que es el método de cualquier clase y dentro de esté le podemos poner cosas como por ejemplo nuestras propiedades y dentro de react.js se les pone **'props'** dentro de esos props van a venir los valores dinámicos que le estamos enviando, en esté caso la que viene será *title*.

```
<h3 className="Media-title">{this.props.title}</h3>
```

Lo que vemos acá es un texto como cualquier otro, para hacer que sea un valor que estamos recibiendo y tiene que estar declarado y dinámico tenemos que ponerlo dentro de 2 llaves, de esté modo estamos asegurando que nos venga el valor como propiedad.

Ya sabes como va la cosa y podemos seguir enviandole propiedades como por ejemplo el author y la imagen. Ahora ya le estamos enviando valores dinámicos a esto. Ahora si nos proveemos una forma de traer datos dinámicos de una api o de algún lugar de nuestra base de datos, podemos ponerle cualquier valor que nosotros queramos.

Ahora que tal si estás propiedades que te llegan y esperan tu elemento no llegan, lo que pasaría es que no se vería el elemento, y eso no esta bien y tenemos que aprender a validarlo para saber que propiedades son las que necesitan nuestros elementos.

Validando tipado en propiedades

¿Que tál si tu componente espera que le llegue una imagen? pero no llega una imagen o que tal si el componente espera que le llegue un author y le llega un author pero ese author no es un texto que diga el nombre del author, sino es su id en la base de datos. ¿Qué tal si los datos nos llegan mal? esto puede romper tu aplicación o no verse como tu deseas y esto tu lo puedes validar y evitar que estos errores se vallan a producción y verlos en desarrollo para que puedas hacer los cambios respectivos a la API, o simplemente que te des cuenta de que estas parseando mal algunos elementos.

Así que podemos validar los tipados de las propiedades para decirle que algo es un texto, algo es una imagen, algo es un número, etc.

Para hacer esto tenemos que instalar una nueva dependencia que desde react@15.5 el validado de propiedades viene como una dependencia aparte, así que vamos a instalarla. instalación: `npm install -D prop-types` Una vez que tengamos los prop-types instalados vamos a poder asignarle eso a nuestra clase media, a nuestro componente media.

Vamos a asignarlos abajo de la case media. `Media.propTypes = { }` Algo adicional que necesitamos es importar lo que acabamos de instalar, así que importemolos desde arriba. `import PropTypes from 'prop-types';` Esta importación la vamos a ocupar abajo, demonos cuenta que el import y el metodo que le acabamos de asignar a nuestra clase Media son diferentes, el método empieza con camel lower case y el modulo que importamos es camel Upper case.

La key propTypes es una propiedad de nuestra clase Media y el import será nuestro modulo que estamos importando. ¿Cuales son las propiedades que tiene nuestra Media? tiene: image, title, author. Los tres keys que enviamos como propiedades son de tipo string, para validarlos tenemos que darle el valor a los keys de nuestro objeto propTypes usando el modulo que importamos. ejemplo

```
import PropTypes from 'prop-types';
Media.propTypes = {
  image: PropTypes.string
  title: PropTypes.string
  author: PropTypes.string
}
```

Obviamente tenemos más tipos, como: bool, number, obj, func, array, etc.

Ahora que tal si nosotros cambiamos alguna propiedad de tipo string la modificamos a un tipo number por ejemplo. Nos daremos cuenta de que el número se imprime porque no hay nigún problema, el número llegá e imprimimos el número, pero la configuración espera que nos llegue un texto. Esto nos causará un warning: propiedad invalida, no se rompe nuestro código pero si nos avisa que el valor que llegá no es el que está esperando.

Esto nos da un valor que nosotros podemos validar, un dato que debería ser otro tipo de dato que si bien en esté caso no rompe la aplicación pero no esta cumpliendo las reglas de la apliación.

Por ejemplo quizas quieras validar textos como por ejemplo de que nos va a llegar el tipo de media que estoy recibiendo ya sea si es audio o es video. Para esto nos va a llegar una propiedad que se llama **type** ejemplo. Esté **type:** podría ser un string, y para ello nos llega un string correctamente. En pocas palabras estamos

cumpliendo las reglas: pero especificamente yo espero que me llegue **video o audio** y que tal si nos llega **videos** o **audios** o otra cosa que no sea las palabras especificas de 'video' o 'audio'.

Estó seguirá funcionando porque sigue siendo un texto no importa que no reaccione mi aplicación como debería ser, nosotros podemos validar con un if el type que nos llegá, pero no nos va a funcionar en caso de que no esté escrito correctamente el string. Para esto nosotros podemos validar el tipo de dato y validar que en lugar de ser un **string** que sea **oneOf([])** el cual es una función pasarle un arreglo de opciones, que lo que hace es escoger al menos 1 de la lista que contenga el array. ejemplo:

```
Media.propTypes = {
  image: PropTypes.string,
  title: PropTypes.string,
  author: PropTypes.string,
  type: PropTypes.oneOf(['video', 'audio'])
}
```

Así vamos a validar que el texto que nos esté llegando ya sea de video o de audio esté muy bien validado. Aunque en esté momento no estamos validando nada en nuestro componente pero esto nos puede salvar la vida. El cúal es muy valioso.

Tenemos que entender que estamos validando el tipo de propiedad que va a ser el titulo, pero no estamos validado que el titulo viene o no viene. Podemos también hacerlo así que vallamos a nuestro componente a media, y decirle a nuestra validación que si queremos que algún valor sea requerido tenemos que decirle que lo sea con **'isRequired'**.

```
Media.propTypes = {
  image: PropTypes.string,
  title: PropTypes.string.isRequired,
  author: PropTypes.string,
  type: PropTypes.oneOf(['video', 'audio'])
}
```

De está manera podemos arreglar múltiples problemas que tengamos en nuestra aplicación porque estó es como el origen de muchas cosas raras que le pueden pasar a tu app, estas esperando valores y llegán de diferente tipo y todo se rompe.

Enlazando eventos del DOM

Recordemos que ReactJS está hecho para crear apliaciones interactivas y altamente poderosas a nivel de su interfaz, pero hasta esté momento no tenemos nada interactivo, esas interacciones son como por ejemplo el click, el paso del mouse sobre un elemento, que pase algo en el navegador cuando ejecutamos alguna opción como usuario como cliente de está interfaz de está apliación.

¿Como enlazo un evento que reaccione a un click dentro de mi compoente? Esto es muy sencillo y es gracias a **on** seguido del evento que queremos eschucar en este caso el click, el cual sería **onClick** y luego lo que queremos ejecutar, en esté caso le enviaremos una función que va a ser **parte** de nuestra **clase Media**.

```
<div className="Media" onClick={this.handlerClick}></div>
```

Usamos this para hacer referencia a algo que está dentro de la misma clase Media. seguido de el nombre de la función separada por un punto.

Otra cosa que vamos a querer hacer al hacer click a ese elemento es poder llamar por ejemplo al title, será que si ponemos {this.props.title} esto funciona?. Esto no funciona porque {this.props.title} no está definido porque esté es un evento de DOM que si bien estamos escuchando no lo estamos enlazando con nuestra clase Media.

Esto lo podemos hacer manualmente por un método que se llamá **constructor**, esté es un método que tienen todas las clases dentro de JavaScript y es una clase que se auto-ejecuta al momento de ser instanciada, es decir que se va ha auto-llamar cuando el componente Media se valla a renderear.

1. Lo que tenemos que pasarle al constructor es decirle que reciba nuestras propiedades
2. Tenemos que hacerle super(props) que indica que está recibiendo las propiedades de su padre.

Lo siguiente que tenemos que hacer enlazar el evento que tenemos del DOM, enlazarlo con nuestra clase, así que tenemos que hacer referencia a nuestro evento usando **'this'** el cual lo estamos enlazando con **this** que es mi mismo componente.

Parte importante:

```
constructor(props) {
    super(props)
    this.handlerClick = this.handlerClick.bind(this);
}
```

Componente Completo:

```
class Media extends React.Component {
  constructor(props) {
    super(props)
    this.handlerClick = this.handlerClick.bind(this);
  }
  handlerClick(event) {
    console.log(event);
  }
  render() {
    const styles = {
      container: {
        fontSize: 18,
        backgroundColor: 'lightgray',
        cursor: 'pointer',
        width: 260,
        border: '2px solid red',
        // padding: 12
      }
```

```
    }
    return(
      <div className="Media" onClick={this.handlerClick}>
        <div className="Media-cover">
          <img
            src={this.props.image}
            alt=""
            width={260}
            height={160}
            className="Media-image"
          />
          <h3 className="Media-title">{this.props.title}</h3>
          <p className="Media-author">{this.props.author}</p>
        </div>
      </div>
    )
  }
}

Media.propTypes = {
  image: PropTypes.string,
  title: PropTypes.string.isRequired,
  author: PropTypes.string,
  type: PropTypes.oneOf(['video', 'audio'])
}

export default Media;
```

Ahora si que vamos a tener disponible en nuestro método handlerClick ese contexto, estamos cambiandole el contexto a mi función para que funcione como deseamos.

Está es la forma de hacerlo con ECMASCRIPT 6 que es como lo vamos a ver escrito dentro de la documentación de react, gracias a ECMASCRIPT 7 lo podemos hacer de una manera más sencilla.

Para ello vamos a comentar el constructor, y vamos a convertir a nuestra función del evento en una arrow function, y dentro de nuestra arrow function nos va a llegar el evento y gracias a esté arrow function que heredan siempre el contexto de su padre, ya tenemos 'this' disponible y vamos a poder imprimir nuestro mensaje de una manera más sencilla y más legible.

Estado de los componentes en React

¿Qué tal si queremos cambiar las propiedades dentro de mi componente? que cuando le de un click cambiemos el author por ejemplo, que cambiemos el valor de la propiedad que estamos recibiendo, digamos que queremos cambiar ese valor y hacer un valor dinámico de la propiedad? **Pues No Podemos**. Porque las propiedades tienen algo muy importante: **Las propiedades son INMUTABLES**. Las propiedades no pueden mutar es decir no pueden cambiar, pero si queremos que algo cambie, ¿como es esto de las aplicaciones dinámicas y ahora no pueden cambiar?

Hay otra forma de cambiar esto y es gracias al estado, el estado si es mutable y si podemos tener valores dinámicos gracias al estado de nuestros componentes, de lo que se trata está clase. Primeramente **tenemos que inicializar un estado** y **para hacerlo** tenemos que volver a darle la bienvenida a nuestro **constructor**, no

nos hemos librado de el tadavía y dentro de nuestro constructor podemos poner a nuestro estado el cual puede tener propiedades nuevas eseciales del constructor, ejemplo:

```
constructor(props) {
    super(props)
    this.state = {
      author: props.author
    }
  }
```

En lugar de usar `${this.props.author}` deberíamos poner `${this.state.author}`. ¿Y si queremos cambiarla, pues vamos a hacer eso? ya que tenemos nuestro click por acá podemos llamar a un método de nuestro componente para modificar esté estado, el del author y ponerle por ejemplo 'Ricardo celis'. Para hacer esto tenemos un método especial para cambiar el estado que se llama **setState()** el cual es una función el cual le vamos a pasar los valores que queramos modificar en nuestro estado, ya que mi estado es un objeto puede tener multiples valores, en esté caso vamos a cambiar el author:

```
handlerClick = (event) => {
    this.setState({
      author: 'Ricardo Celis'
    })
  }
```

Si nosotros probamos nuestro click veremos que el resultado cambió, y esto es mágico además es una de las cosas de como en React vamos a manejar datos dinámicos, datos que vamos a estar calculando cuando un evento ocurra en el navegador, como por ejemplo que queremos manejar el responsive o cambiar algunas cosas, pues lo hacemos de esa manera.

Usando el Constructor es gracias a ECMASCRIPT 6. ¿Qué tal si lo resumimos con ECMASCRIPT 7?

La cual es muy sencillo como poner lo siguiente:

```
 state = {
    author: 'Leonidas Esteban'
  }
```

¿Qué más trae mi componente? A eso que trae de más de más en el componente se le llama el ciclo de vida, que ocurre cuando el componente se va a pintar en la pantalla que ocurre antes, que ocurre despues, hay una forma de saber si el componente se pinto en la pantalla, por supuesto que la hay y ha eso se le llamá el ciclo de vida de un componente.

Ciclo de vida de los componentes

Ya aprendimos mucho acerca de React, ya aprendimos a crear componentes, a ponerle estilos, a mandarle propiedades, a ponerle un estado, a manejar eventos y ahora tenemos que aprender muchisímo más y algo

muy importante acerca de react es que aparte de ponerle propiedades y a manejarle un estado.

Los componentes tienen un ciclo de vida, que es básicamente {como entran en escena, como se van de escena, algunas actualizaciones que tienen los componentes y algunas de estas cosas las podemos capturar gracias a react}, y de eso se trata está clase.

El ciclo de vida de un componente se **divide en 3 partes**

- El **Montado**
- La **Actualización**
- El **Desmontado**, y desde **React16** también tenemos el **Manejo de errores**

Empecemos por El Montado: En el montado tenemos varios métodos que podemos manejar al momento de que tenemos un componente en react y el montado es está parte de cuando vamos a renderizar un componente cuando este va ha entrar en escena dentro de tu apliación de eso se trato el montado. métodos:

- El constructor que es un método de cualquier clase de javscript, esté método se va a llamar una vez que queramos instanciar una clase, ahí podemos ver que podemos poner como el estádo inicial, podemos hacer **bind** de enventos. Lo único que se recomienda hacer es ponerle estado inicial o **bindear** enventos o ponerle cualquier propiedad a **this** dentro de está clase, como por ejemplo: this.nameComponent, nombre del componente y ponerle algun nombre al componente para luego llamarlo como 'this.algo' en cualquier otro lado del componente, no se recomienda hacer nada más que para eso existen los pasos del ciclo de vida del componente. **Recordemos que esté método es llamado antes de que el componente sea MONTADO**

- **ComponentWillMount:** Es el método que se va a llamar inmediatamente antes de que el componente se valla a montar, esto quiere decir que ya no hay un problema todavía en el componente, el componente siguiendo su ciclo de vida, no ha existido un problema dentro del constructor. Seguimos al **componentWillMount**. Por ejemplo si tenemos alguna propiedad y queremos setear algún estado con base a esas propiedades, utilizamos un setState ahí probablemente no lo necesite podría hacerlo. Lo que te recomienda React es que no hagas un llamado a una API o cualquier subscripcion a un evento, porque esté componentWillMount no lo vamos a ver todavía pero también se ejecuta cuando queremos hacer un renderizado de componentes dentro del servidor, por ejemplo en el Servidor no tienes al DOM para hacer un bind de envetos, como no tienes que hacer un *addeventlistener* eso no lo tienes, por eso no te recomiendo que en ese momento no hagas un bind de enventos.

- **Render:** El render es donde vamos a poner todos los elementos; todo el JSX, toda la estructura de nuestro componente y adicionalmente que tienes todo el contenido dentro del componente podemos procurar algún dato como por ejemplo: si te llegarán 2 propiedades como por ejemplo: name y lastname podrias juntarlos y tenerlo en una sola variable o podrías hacer ese cálculo de juntarlo como si fueran 2 textos dentro del return, esa lección va más de tu parte como desarrollador. Contiene todos los elementos a renderizar y la estructura de nuestro componente, luego de estó mi componente ya se a ver dentro de la pantalla.

- **ComponentDidMount:** Es el componente que se llama luego de que el componente se ha montado y como ya te comente en este momento el componente ya está en pantalla, ya está dentro de tu navegador esto quiere decir que ya que está en pantalla dentro del navegador ya podemos utilizar las apis del navegador como por ejemplo: **hacer un setTimeout, hacer un setInterval, hacer un window.addEventListener** o si quieres llamar datos hacia una API esté sería el lugar correcto.

Actualización: Pasamos a la siguiente parte del ciclo de vida que es la actualización, que hasta acá tenemos el primer pedazo que es cuando el componente recién va a entrar en escena y cuando el componente recién entro en escena, ¿pero que sucede si esté componente recibe nuevos datos? Es aquí donde entramos a la parte de ACTUALIZACIÓN.

Partes o métodos de la actualización:

- **ComponentWillReceiveProps:** Esto es como del futuro, ¿El componente recibirá proiedades? Esté es el método que va a recibir las nuevas propiedades de mi componente entonces es aquí que podemos validar las propiedades que yo tenía como mi componente. ¿Aquí puedo validar las propiedades que mi componente son iguales que las nuevas propiedades que tengo en mi componete? Y aquí podemos setaer algún estado y puedo hacer eso y sobretodo puedo maniobrar mucho más con los componentes, **sirve para actualizar el estado con base a las nuevas propiedades**

- **shouldComponentUpdate:** Esté método es super importante al mejorar el rendimiento de nuestra apliación, porque ¿que tal si las propiedades que le llegan a mi componente son las misma propiedades que ya tenía? Para esté caso yo no necesito hacer un re-Render y acá en el componente *shouldComponentUpdate* podemos validar esto. Es decir ¿las propiedades que ya tenía son las mismas que voy a tener? Bien entonces no hagas re-Render simplemente en esté componente devolvemos false y el componete no vuelve a hacer un re-Render.

- **componentWillUpdate:** Estó quiere decir que si pasó el *shouldComponentUpdate* si devolvió true, va a pasar al siguiente, esto quiere decir que el componente se va ha actualizar y esto significa que se va a re-Renderizar va a volver a lanzar al método Render.

- **Render:** Aquí tenemos otra vez el llamado a nuestro método render y ha estó se le llamaría un re-Render.

Ahora que ya sabemos como se ha actualizado tenemos como final de está parte de actualizaciones a:

- **componentDidUpdate:** Que siginifica que el componente ya se actualizo, así como el didMount, pero ahora el didUpdate, aquí esté componente se re-Renderizo.

Siguiendo estó de que ya tenemos la parte de cuando el componente entra en escena, se actualiza en la escena, ¿Qué pasa cundo el componente se va de la escena? o lo queremos quitar con un validación, sigue el DESMONTADO de mi componente que es como esté componente se va de la escena.

- **componentWillUnmount:** que es el método que se va a lanzar inmeditamente antes de que el componente se valla de la escena. Digamos que tenemos un reproductor de video, tenemos un botón de play y uno de pausa, qué es como que vienen en el mismo lugar, play y pause, que si le doy click a uno me aparece el otro. Ahí vamos a ver que esté método estaría llamandose porque los componentes se estaría desmontando tanto el play como el pause.

Adicionalmente como lo especificamos arriba, desde React@16, tenemos el **Manejo de errores* lo cuál es muy interesante porque nos va a poder prevenir que toda nuestra apliación se rompa si algún componente tiene algún problema, entonces podemos manejar los errores dentro del componente de su ciclo de vida. Siempre hemos podido manejar errores pero desde que es parte del ciclo de vida podemos prevenir que estos errores rompan toda la apliación y al mismo tiempo mandarle un servicio externo como un API o Centry que es para enviar errores y reportarlos, para decir esté componente falló porque de repente no le llegaron ciertas

propiedades porque estaba mal escrito en cierta parte, esto es lo que nos dirá nuestro método del manejo de errores, que se llama: *componentDidCatch*

- **componentDidCatch:** Cuando encontremos un error esté ya encontro el error no va a prevenir que el componente no tenga errores, si el componente falla este es el componente del ciclo de vida que va a ser llamado, si algo tendrá un problema de renderizado, entonces se lanzará esté componente.

Algo muy interesante es que el manejo de erroes ocurre de padres a hijos, si queremos que un componente sea prevenido de sus errores, tengo que envolverlo en algo que maneje esos errores, quiero que el padre si nos diga: oye tu hijo falló y así es como va a funcionar.

Listas en ReactJS

Una de las necesidades que tiene nuestra aplicación es renderizar multiples elementos, actualmente tenemos nuestro componente de media, pero nuestro componente de media no va a vivir solo porque tiene que estar acompañado de sus hermanos, otros media que van a estar aquí en su derecha, entonces como rendereamos multiples elementos apartir de unos datos, para empezar ya no vamos a mandarle datos a este componente por medio de propiedades de está manera, así como harcodeadas como si esta fuera base de datos, necesitamos una forma que contenga todos los datos de los medias, para eso en este curso tenemos un archivo json que va a ser usado como nuestra base de datos, el cual se llama 'api.json', ahora lo que haremos será importar esté archivo json al lugar donde la ocuparemos en este caso sería index.js. Como la extensión no es JS toca ponerle el nombre de la extensión.

Ahora puedo pasarle estos datos a mi Media, pero saben que yo quiero que mi Media no viva solo, así que hagamos otro componente que contenga toda la playlist y que esté componente si que renderize Media, para eso tenemos que crear un nuevo componente, vamos a crear un componente playlist que es el componente que va a renderizar nuestras Medias.

playlist component

```
import React from 'react';
import Media from './Media.jsx';
import './playlist.css';

class Playlist extends React.Component {
  render() {
    const playlist = this.props.data.categories[0].playlist;
    console.log(playlist);
    return (
      <div className="Playlist">
        {
          playlist.map((item) => {
            return <Media {...item} key={item.id}/>
          })
        }
      </div>
    )
  }
}
```

```
export default Playlist;
```

Ahora ya no vamos a renderizar Media dentro de nuestro index, ahora el que se renderizará será Playlist, para eso tenemos que importarlo, el cual quedará de la siguiente manera:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Playlist from './src/playlist/components/playlist.jsx';
import data from './src/api.json';

// ReactDOM.render(que voy a renderizar, donde lo haré)
const app = document.getElementById('app');
ReactDOM.render(<Playlist data={data}/>, app);
```

Si el componente no tiene nada dentro de si mismo, lo importante es que lo cierren en sí mismo. La playlist necesita ciertos datos, los cuales vamos a enviarselos por medio de una propiedad.

De está manera voy a tener datos dentro de mi playlist que me llegan como propiedades.

Componentes Puros y Funcionales en ReactJS

Hasta esté momento hemos ocupado los componentes más clásicos de react pero hay otras 2 formas de hacer componentes: **componentes puros y componentes funcionales** ¿Cuál es la diferencia y cuando debo usarlos?

Así como trabajamos con Component, react nos provee de otro método que se llamá *PureComponent*. ¿Cuál es la diferencia? Recordemos que cuando vimos el ciclo de vida, había una parte del ciclo de vida de la actualización que se llamaba shouldcomponentUpdated, pues PureComponent tiene shouldComponentUpdated ya asignado es decir si ha esté componente no se le actualizan las propiedades, no tengo que validar a mano con shouldComponentUpdate que eso ocurra porque PureComponent lo va a ser por mi, está es **la única diferencia entre un component y un purecomponent**.

No es para menos pues si queremos mejorar el rendimiento de la apliación pensando en el re-Render de está misma y un componente se puede actualizar multiples veces como probablemente sea nuestro componente de Media tu debería utilizar un PureComponent y el resto de cosas las tenemos normal, tenemos el método render y nada ha cambiado.

Componentes Funcionales: Las reglas cambian un poco cuando queremos hacer componentes funcionales. **Un componente funcional** recibe el nombre porque **es una función** ejemplo: de componente Funcional

```
function Playlist(props) {
  return(
    <div>
      playlist
    </div>
  )
}
```

Hacer los componentes de está manera es mucho más sencillo que escribir el método render, podemos hacer un componente que sea solo un render, solo va a tener la AI, lo siguiente es mucho más fácil de probar, también podemos recibir propiedades. Sobre todo es más fácil de escribir y probar Pero esté componente Funcional no tiene un Ciclo de vida, porque no hay forma de poner los métodos de componenwillMount, etc.

Si esté componente por ejemplo llamá a un método como por ejemplo un 'click', podríamos hacer otra función y acá hacer lo que haga el click, aunque esto no es una buena práctica ya que no es el enfoque que vamos a utilizar, sería mucho más sencillo que como se maneje el click no dependa de esté componente funcional, que más bien sea una propiedad que nos llegue, para que un componente que tenga estado si maneje, el click. Estó de cuando utilizar componentes puros o funcionales con estado completo y todo eso es una metodología que vamos a profundizar en el siguiente tema.

Ahora a nuestro playlist como no cambia su estado, y solo es despligue de AI. **Si no nos hace falta un ciclo de vida, lo más adecuado es hacer un componente funcional**.

```
function Playlist(props) {
  const playlist = props.data.categories[0].playlist;
  console.log(playlist);
  return (
    <div className="Playlist">
        {
          playlist.map((item) => {
            return <Media {...item} key={item.id}/>
          })
        }
      </div>
  )
}
```

Al final del día esté componente funcional que se escribe mucho más corto, se va a transformar en el componente que era anterior, el cual extendía de component, pero acá nos queda mucho más fácil de leer y sabemos que lo único que va a hacer este componente es renderizar AI.

SMART Componentes

Nuestra apliación va ha ser muy grande y cada parte de la apliación va a tener un responsabilidad totalmente diferente, algunos componentes van a ser con estado, algunos componentes van a ser Puros, algunos serán funcionales, ¿Pero cuando utilizarlos? ¿Como saber si un componente va a tener un estado o no? ¿Acaso todos los componentes podría tener un estado? la verdad es que sí, pero ¿Debería tenerlo?

Aquí es donde Adan Abramov nos enseña la diferencia.

¿De que se trata el smart y Dumb component? se trata de **Como se ve VS Como se hace**

El Dumb sería como se ve un componente y lo que hace mi componente es el smart

Hay muchas otras manera de llamar a estos smart y dumb components, otras formas de llamarlos:

Con estado: "Smart", "Statefull" Sin estado: "Dumb", "Pure"