Curso de React

Tabla de Contenido

¿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.

↑ volver al inicio

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.

↑ volver al inicio

Particularidades de React

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.

↑ volver al inicio

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.

↑ volver al inicio

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 webpack

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%

↑ volver al inicio

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.

↑ volver al inicio

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.

↑ volver al inicio

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 Javascript 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.

↑ volver al inicio

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.

↑ volver al inicio

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.

↑ volver al inicio

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.

↑ volver al inicio

Componentes

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.

↑ volver al inicio

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**

↑ volver al inicio

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.

↑ volver al inicio

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**

↑ volver al inicio

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.

↑ volver al inicio

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:

↑ volver al inicio

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.

↑ volver al inicio

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*

↑ volver al inicio

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.

↑ volver al inicio

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.

↑ volver al inicio

Componentes Puros

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.

↑ volver al inicio

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.

↑ volver al inicio

División de Components

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:	**Sin estado:**
Smart	Dumb
Statefull	Pure
Fat	Skinny
Container	Presentational o Component

Según explica Adan no es exactamente lo mismo pero tienen el mismo espiritu, que dividen responsabilidades.

↑ volver al inicio

Dumb or Presentational

1. Puede contener Smart components u otros componentes de UI
2. Permiten composición con {props.children}
3. No depende del resto de la apliación
4. No especifica como los datos son cargados o como mútan
5. Recibe datos y callbacks solo como propiedades
6. Rara vez tiene su propio estado
7. Están escritos como componentes funcionales a menos que necesiten mejoras en el performace.

↑ volver al inicio

SMART or Containers

1. Concentrado en el funcionamiento de la apliación
2. Contiene componentes de UI u otros componentes
3. No tienen estilos
4. Proveen de datos a componentes de UI u otros contenedores
5. Proveen de callbacks a la UI
6. Normalmente tienen estado
7. Llaman Acciones
8. Generados por higher order components

Es como si fuera el manager de la apliación o el titiretero que va a estar viendo como las cosas estan funcionando, el componente inteligente sabe que hay dentro de los otros componentes hijos, ya que va a manejar como funciona la interfaz.

↑ volver al inicio

¿Porque es importante esta división?

Por 2 puntos importantes que comom frontend me interesan muchisímo:

1. Separación de Responsabilidades

Como se ve VS como funciona. Por ejemplo un componente Puro o funcional de UI, un componente Tonto sería nuestra vista de nuestro MVC, solo la vista, solo como se ve, Mientras que un componente inteligente o Container sería nuestro Controlador de nuestro modelo view controller. Así como una convención, Toda está métodologia es una convención que deberíamos utilizar.

2. Mejora la capacidad de reutilizar componentes

Es por eso que hacemos esta separación de responsabilidades, porque si hacemos que un componente sea nadamas el botón y esté botón no maneje su propio click, que hace su click dentro del botón, podría yo ocupar ese botón en otro lado de la aplicación y que en ese otro lado de la apliación cuando demos click, ese click haga otra cosa. En algún momento ese click abre un modal y en otro lado donde se vuelva a utilizar esté mismo botón ese click lo que hagá sería desplegarme una lista de elementos, entonces se ven iguales pero hacen cosas diferentes, gracias a sus contenedores. Por eso es tan importante y apartir de ahora nuestra apliación va a ser hecha con está **metodología** de **Smart and Dumb Components** o **Presentational and Containers**.

↑ volver al inicio

Composición de Componentes

Ahora aprendamos como funciona la composición de componentes y como ya empiezan a funcionar nuestros componentes funcionales(componentes Tontos) y como podemos heredar esto o componer funcionalidades dentro de nuestros componentes. El mini proyecto que haremos será para un pack de iconos que nos va a servir, cuando hagamos el reproductor de video de nuestro proyecto 'platzi-video'. Para eso vamos a crear una nueva carpeta que se llamé 'icons' y dentro de nuestros icons vamos a poner una carpeta para nuestros componentes.

Apartir de ahora donde tengamos alguna carpeta 'components' nos vamos a referir que aquí adentro solo existan componentes PUROS O FUNCIONALES. Si creamos una carpeta *CONTAINERS* ahi van a estar los componentes con estado o Componentes inteligentes.

Para ello haremos un componente de icono para nuestro play del reproductor

```
import React from 'react';
function Play() {
  return (
    <div>Play</div>
  )
}
export default Play;
```

Ahora lo que tenemos que hacer es importar los iconos que son archivos svg que simplemente son iconos de como se deben ver los botones. Para ello vamos a copiar el codigo fuente de este archivo svg dentro de mi componente play y ver que tal nos queda el play y así volvemos una svg a un componente. Si ahora intentamos imprimir nuestro Play lo podemos lograr renderizando en el navegador.

Ahora nosotros queremos que nuestro play sea configurable, ponerle un tamaño a nuestro icono, poder pasarle algunas propiedades extras para personalizarlo, para que nuestro componente incono se pueda exponer en cualquier parte de la apliación, no solo en nuestro reproductor que estamos creando. Pues podríamos pasarle propiedades a nuestro icono.

Otra cosa que nos va a pasar es que vamos a tener muchos componentes de iconos y todos van a ser básicamente lo mismo, van a recibir un tamaño y un color que es lo minimo que necesitaría nuestro icono de play, así que, que tal si hacemos que esa configuración solo se haga en un lugar para no configurar: play, pause, fullscreen, etc. Entonces crearemos un componente de icon.

Vamos a crear nuestro componente funcional, el cual recibirá las propiedades y va a retornar que va a tener nuestro icono que va a ser un wrapper, un contenedor de nuestro botón de play. Le podemos decir que lo que contenga el icono sea otro componente, en este caso podriamos hacerlo así:

```
import React from 'react';
import Play from './play.jsx';
function Icon(props) {
  return (
    <div>
      <Play />
    </div>
  )
}
export default Icon;
```

Pero estó no es escalable porque solo estamos importando play, pero que pasa cuando estamos exportando Volumen, o FullScreem, etc. Podríamos hacer un if o un switch por cada uno de estos para que funcione con todos mis iconos. **Pero hay un forma más sencilla que sería la composición de elementos dentro de ReactJS**. Si queremos pasarle cualquier cosa a mi icono sería que lo que yo ponga dentro de mi container sea {props.children}. Y ahora cada vez que importemos un icono me va a mandar a su hijo, entonces yo en mi play podría importar a icono para que extienda sus poderes.

Entonces lo que haría mi play sería ponerme aquí un Icon y dentro de lo que haiga de ese Icon. **Esto señores es el children de el Icon**. Esto es lo que va a imprimir mi propiedad Icon y lo va a imprimir dentro de ese div que todavía no hace nada.

Pero todas las capacidades que tenemos en cada incono las cuales estan hardcodeadas, esas capacidades ya se las podemos poner como configuración dentro de mi icono, para que así podamos darles tamaños diferentes a cada elemento de icono que vallamos a utilizar y no solo dependan de play, de pause o de alguno particularmente. Podemos quitarle el svg y title, y solo dejarle el **path** que es lo que diferencía como se ve nuestro play, asi es como funcionan los svgs.

```
import React from 'react';
import Icon from './icon.jsx';

function Play() {
  return (
    <path d="M6 4l20 12-20 12z"></path>
  )
}

export default Play;
```

Ahora nuestro icono si tiene que manejar más responsabilidades el viewboxing y el tamaño de mi svg. Para empezar acá le estamos dando un path y mi icono hasta esté momento es un div, cambiemoslo por `<svg>` para que adentro tenga su path y acá configurar de una vez como es que se tiene que ver.

```
import React from 'react';

function Icon(props) {
  return (
    <svg
      fill={props.color}
      viewBox="0 0 32 32"
      width={props.width}
      height={props.height}
    >
      {props.children}
    </svg>
  )
}

export default Icon;
```

Aunque podemos mejorar como se hace la composición de esto, haciendo una constate, que me traega las propiedades del svg.

Ahora vamos a ir a nuestro playlist donde tenemos nuestro play y a nuestro play le vamos a pasar esas propiedades:

```
import React from 'react';
import Media from './Media.jsx';
import './playlist.css';
import Play from '../../icons/components/play.jsx';

function Playlist(props) {
  const playlist = props.data.categories[0].playlist;
  console.log(playlist);
  return (
    <div className="Playlist">
```

```
        <Play
          size={50}
          color="red"
        />
          {
            playlist.map((item) => {
              return <Media {...item} key={item.id}/>
            })
          }
        </div>
    )
  }
```

Ahora estás propiedades se están llendo a mi componente de Play que a su vez utiliza Icon, pero Icon es el que esta recibiendo propiedades pero Play no recibe ninguna propiedad, para eso vamos a heredar las propiedades que le llegan a Play para que Icon las pueda utilizar, de la siguiente manera:

```
import React from 'react';
import Icon from './icon.jsx';
function Play(props) {
  return (
    <Icon {...props}>
      <path d="M6 4l20 12-20 12z"></path>
    </Icon>
  )
}
export default Play;
```

Estó será lo mismo con los demás Iconos.

```
import React from 'react';
import Icon from './icon.jsx';

function Pause(props) {
  return (
    <Icon {...props}>
      <path d="M4 4h10v24h-10zM18 4h10v24h-10z"></path>
    </Icon>
  )
}

export default Pause;
```

En el Playlist.jsx Tenemos que agregarlos, ya que no los estamos iterando, solo los estamos llamando

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

```
import './playlist.css';
import Play from '../../icons/components/play.jsx';
import FullScreen from '../../icons/components/full-screen.jsx';
import Volume from '../../icons/components/volume.jsx';
import Pause from '../../icons/components/pause.jsx';

function Playlist(props) {
  const playlist = props.data.categories[0].playlist;
  console.log(playlist);
  return (
    <div className="Playlist">
      <Play
        size={50}
        color="red"
      />
      <Volume
        size={50}
        color="blue"
      />
      <Pause
        size={70}
        color="orange"
      />
      <FullScreen
        size={200}
        color="green"
      />
        {
          playlist.map((item) => {
            return <Media {...item} key={item.id}/>
          })
        }
      </div>
  )
}
export default Playlist;
```

↑ volver al inicio

Estructura del proyecto con Containers y UI components: layout de nuestro sitio web

Ahora que ya sabemos como va a funcionar nuestro proyecto, ya podemos dividir las responsabilidades y que componente utilizar dependiendo cada una de esas responsabilidades, veamos cual va a ser la estructura que va a tomar nuestro proyecto de platzi video.

Primero veamos el diseño y veamos que tenemos 2 columnas, pero ya en vista en parte de componentes tenemos que ponerle un nombre y tenemos que jerarquizar, pero antes de eso weboack nos pide un entrypoint, para ello iremos a nuestro archivo components.md donde ya tenemos organizados nuestros componentes.

Webpack nos pide un entry point de como se va a llamar nuestro proyecto, en esté caso sería nuestra home que iría directa a webpack, que sería la home de platziVideo, porque en platzivideo tenemos multiples entrypoints, luego de esa home que sería ese entrypoint para webpack importariamos si al componente de Home, donde Home es un componente, va a ser nuestro componente principal, va a ser nuestra página que va a envolver todo lo que estamos haciendo en esté curso, y nuestra página debería ser un componente inteligente. Luego de estó si queremos ponerle una clase o un estilo a nuestro container principal, que si vamos al diseño nuestro container principal sería un `<div>` que tiene todo el contenido de la página.

Pero si queremos ponerle estilos a este div no podemos hacerlo porque un Container no debería de tener estilos, podrían ponerlos pero no deberían, entonces le vamos a crear un componente de layout, solo para ponerle estilos a la home, a ese div principal, también vamos a aprovechar para ponerle los estilos al body y ha esto que sería los básicos por página. Luego está página se divide en 2 columnas, izquierda y derecha. Al componente de la izquierda que es un componente de UI, para esté componente dejaremos al estudiante a que haga la jerarquía de este componente ya que en el curso solo haremos el de la derecha. En la derecha vamos a tener nuestras categorias, dentro de las categorias tenemos un categoria y una categoría tiene un playlist y un playlist tiene un Media, actualmente tenemos esto: {playlist y Media}, vemos que el Media tiene que ser de UI y tiene que ser Puro, nos falta hacer {categories y category} para que se impriman las diferentes filas y estas estan contenidas dentro de nuestra Home, nuestra Home no existe, solo existe nuestro home: pero de entrypoint, así que empecemos a hacer esto.

Luego de esto tenemos buscador que lo vamos a ver luego, y a nuestro Modal que adentro vamos a tener nuestro reproductor de video donde estó se pone como que más interesante, los cuales los veremos en su devido momento.

Primero vamos por está parte: el entrypoint, la página y despues hacemos las partes de las categorias, haremos esté pedacito en esta clase.

Ahora vamos a empezar con nuestro entrypoint; para lo cual vamos a empezar a modular nuestro proyecto y acá pondremos a nuestro entry en una carpeta especial de entrypoints.

```
entry: {
    'home': ['babel-polyfill', path.resolve(__dirname, 'src/entries/home.js')],
  },
```

↑ volver al inicio

Portales

Hasta ahora ha vivido dentro de un tag HTML que ya teníamos puesto dentro de nuestro html que se llamá app, recordemos que esto está siendo renderizado por react-dom y su método render, esta buscando ese tag y lo coloca ahí. ¿Pero que ocurre si queremos poner contenido que no este dentro de app? Para esto react-16 trae una nueva caracteristica que se llaman portales que es la forma de que otro componente viva dentro de otro div que no sea apliación.

En nuestro caso de uso para la aplicación que estamos construyendo es construir nuestros modales, nuestra ventana de modal, para ello vamos a crearle un contenedor que renderice nuestro modal, esto nos sirve para que nuestro modal se renderice en otro lugar de nuestro HTML, porque nos podría dar algún problema con

los estilos, aunque se podría manejar dentro de app, pero lo mejor es tenerlo en un lugar aparte del otro contenedor.

Para ello al no ser 'app' nuestro único div de nuestra apliación vamos a cambiar los nombres para ser más explicitos con nuestro código el cual quedaría así.

```
<body>
  <div id="home-container"></div>
  <div class="" id="modal-container"></div>
  <script src="http://localhost:9000/dist/js/home.js"></script>
</body>
```

de igual manera tenemos que leer nuestros elementos del DOM con estos nuevos nombres.

Para crear nuestro portal iremos a nuestro Componente Inteligente o contenedor(container) en esté caso es el Home, tenemos que darle una manera de que renderé ese contenido, pero como vamos a crear un portal tenemos que renderearlo dentro de su propio componente. Esté componente será un componente Inteligente porque va a manejar como nuestro Modal va a funcionar. Inclusión del Portal en nuestro Componente Principal de la Aplicación:

```
render() {
    return (
        <HomeLayout>
        <Related />
          <Categories categories={this.props.categories}/>
          <ModalContainer>
            <h1>Esto es un Portal</h1>
          </ModalContainer>
        </HomeLayout>
    )
  }
```

Ahora crearemos esté componente:

```
import React from 'react';
import { createPortal } from 'react-dom';
class ModalContainer extends React.Component {
  render(props) {
    // createPortal(que voy a renderizar, donde?)
    return createPortal(
      this.props.children,
      document.getElementById('modal-container')
    )
  }
}
export default ModalContainer;
```

Esté componente será inteligente porque es un contenedor el cual manejará estado. En esté componente tenemos que retornar un Portal y ese portal es una function que nos da React que se llamá **createPortal()** está función es una función que renderiza Al, y todo lo que se renderiza en el Dom es atraves de 'react-dom' y la función es parte de react-dom. Tal cual como el método *render()* de react-dom recibe los mismo párametros: (que voy a renderizar, donde?)

Lo que vamos a renderizar en nuestro portal es lo que le mandemos en esté caso nuestro Modal puede contener muchas cosas, pero nuestro Modal tiene que ser reutilizable, así que este modal puede contener cualquier cosa aparte de lo que le mandemos en este ejemplo, para ello haremos que renderé a los hijos, esto quiere decir que cualquier cosa que este contenida dentro de el Tag `<Modalcontiner>Aqui van los hijos...</ModalContainer>` con esto hacemos que el componente sea reutilizable, y A donde lo enviaremos será a nuestro nuevo Tag HTML que es `<div class="" id="modal-container"></div>` Ahora ya podemos importarlo dentro de nuestra Home y manipularlo desde ahí. Si inspeccionamos el código de nuestro página podremos ver que el contenido se ha renderizado dentro del div de modal-container, no está dentro del home-container apesar de que nuestro home está renderizado dentro de HomeContainer. Pero gracias a los portales es que podemos mandar esté componente a otro lado.

Ahora que ya aprendimos a hacer nuestro Portal vamos a aprender a enviar propiedades en la seguiente sección.

↑ volver al inicio

Modal

Ya sabemos que los portales nos sirven para renderear contenido dentro de otro elemento HTML que ya pre-exista en nuestra apliación de React y no lo hemos utilizado para crear nuestro portal que va a utilizar nuestro Modal, pero no hemos echo nuestro Modal y en esta sesión aprenderemos a hacer y a enviar y recibir propiedades dentro de nuestro Portal.

1. Crearemos primero nuestro Modal, el cual también recibirá contenido dinámico.

```
import React from 'react';
import './modal.css';
function Modal(props) {
  return (
    <div className="Modal">
      {props.children}
      <button onClick={props.handleClick}>Click</button>
    </div>
  )
}
export default Modal;
```

2. Lo importamos dentro de nuestra Home igual como su contenedor `import Modal from '../../widgets/components/modal.jsx';` Podemos observar dentro de Home que React se ve muy declarativo y esto es una de la filofica de React, es que la app sea muy legible.
3. Ahora Agregaremos los estilos para que se ve más como un Modal. Nuestro modal es condicionado para cuando interactue con el usuario y para ello vamos a enviarle las propiedes. Recordemos que los

componentes Puros o funcinales no deberían manejar sus propios eventos, los cuales deberían venir por propiedes.

```
<button onClick={props.handleClick}>Click</button>
```

5. Ahora iremos a nuestro Home y a nuestro Modal vamos a enviarle esa propiedad la cual la estamos consumiendo dentro del Modal `<Modal handleClick={this.handleCloseModal}>`. Donde vamos a tener más clicks serán dentro de nuestro componente inteligente en esté caso, ese componente es Home. En home crearemos nuestras funciones las cuales serán enviadas por propiedes a nuestros componentes Puros.

```
class Home extends React.Component {
  state = {
    modalVisible: true,
  }
  handleCloseModal = (event) => {
    this.setState({
      modalVisible: false,
    })
  }
}
```

El estado del Modal es true y para renderizar nuestro componente crearemos una condicional, con su estado y si se cumple entonces lo renderea.

```
{
  this.state.modalVisible &&
  <ModalContainer>
    <Modal handleClick={this.handleCloseModal}>
      <h1>Esto es un Portal</h1>
    </Modal>
  </ModalContainer>
}
```

Esto es solo si la condición se cumple '&&' si queremos hacer un if podemos hacerlo como if-ternario usando '?' para if y ':' para else.

Nuestro Modal aparece por defecto pero nosotros queremos que solo aparecesca al darle click a un Elemento img, el cual es nuestro Componente Media.

1. Nuestro estado inicial del modal será 'false'
2. Vamos a pasarle una función que haga que el modal se prenda.

```
handleOpenModal = (event) => {
    this.setState({
```

```
      modalVisible: true,
    })
  }
```

3. Vamos a recibir la propiedad en Media.

```
<div className="Media" onClick={this.props.handleClick}>
```

4. Ahora tenemos que enviarle la propiedad desde las categorias, porque es lo más cercano que tenemos al padre todo que es nuestro contenedor.

```
return (
    <div className="Categories">
      {
        props.categories.map((item) => {
          return (
            <Category
              id={item.id}
              {...item}
              handleOpenModal={props.handleOpenModal}
            />
          )
        })
      }
    </div>
  )
```

5. Ahora tiene que pasar de la Category al Playlist y del playlist al Media.

```
function Category(props) {
  return(
    <div className="Category">
      <p className="Category-description">{props.description}</p>
      <h2 className="Category-title">{props.title}</h2>
      <Playlist
        handleOpenModal={props.handleOpenModal}
        playlist={props.playlist}
      />
    </div>
  )
}
```

6. Ahora ya la heredo la playlist y de la aquí se la enviaremos a Media.

```
function Playlist(props) {
  return (
    <div className="Playlist">
        {
          props.playlist.map((item) => {
            return (
              <Media
                {...item}
                key={item.id}
                handleClick={props.handleOpenModal}
              />
            )
          })
        }
      </div>
  )
}
```

↑ volver al inicio

Manejo de Errores

A lo largo de las aplicaciones dentro de ReactJs seguramente algún componente va a fallar porque no le van a llegar datos. Porque le llegarán datos de más. o por algún dato que le valla a llegar valla a ser diferente al que se está esperando, y por más que estamos validando estó con propiedes y las propiedes estén diciendo que el dato es diferente y esto hará que la apliación se rompa y no se vea nada en el navegador.

Desde React-16 tenemos una forma de capturar esos errores y podemos mostrar al usuario que existe un error dentro de la página y esto lo podemos hacer desde el propio componente, desde su ciclo de vida. Esta parte del ciclo de vida del componente se llamá **componentDidCatch(error, info)** este método nos proporciona 2 párametros: el error y la información. Lo que tenemos que hacer con estó es si tenemos algún servicio como centry es enviarlo para que luego ese error sea monitoreado y luego con el equipo de desarrollo poder solucionarlo.

Otra cosa que podemos hacer aquí es que el frontend ya que tiene una forma de capturar esté error, el frontend te puede mandar un error dentro de la página. Entonces cuando ocurra un error dentro de esté componente y con cualquiera de sus hijos, vamos a setear un estado y si el estado es verdadero vamos a rederear en la página que hubo un error.

```
state = {
    handleError: false,
  }
componentDidCatch(error, info) {
    // Envía este error a un servicio como Sentry
    this.setState({
      handleError: true,
    })
  }
```

Ahora validaremos ese estado en nuestro render() y para validarlo podemos escribir lo que sea dentro de javascript

```
render() {
  if (this.state.handleError) {
    return <p>Ohhh hay un error :(</p>
  }
}
```

Para ello provocaremos un error mandandole propiedades erroneas a nuestro componente y de esté modo provocaremos el error, el cual hará que nos retorne el mensaje de arriba.

Tal vez te preguntes, como hago para heredar al menos este componentDidCatch para devolver un error si todo esta bien y un error si todo esta mal. Pues podemos crearnos un componente que maneje estos problemas y nos ayude dentro de nuestra aplicación.

Vamos a crear un componente que se encargue de hacerlo. Si vamos a ocupar el componentDidCatch tenemos que usar un ciclo de vida de componente.

```
import React, { Component } from 'react';
import RegularError from '../components/error.jsx';

class HandleError extends Component {
  state = {
    handleError: false,
  }
  componentDidCatch(error, info) {
    // Envía este error a un servicio como Sentry
    this.setState({
      handleError: true,
    })
  }
  render() {
    if (this.state.handleError) {
      return <RegularError />
    }
    return this.props.children
  }
}

export default HandleError;
```

Ahora que ya tenemos nuestro componente para menjar los errores procedemos a importarlo en nuestro componente de Home. `import HandleError from '../../error/containers/handle-error.jsx';` Algo muy interesante es que en nuestro componente Home ya podemos envolver nuestra UI con el manejador de errores, de esté modo si fallamos en recibir los datos, el manejador de errores nos retornará el Error cuando fallé, en este caso nuestro Error es un UI aparte que se llama 'regular-error' el cual es un pure component el cual solo nos retorna una UI con el mensaje del error, pero nosotros podemos personalizar

como quieramos el error y agregar otros tipos de errores. Recordemos que esto es gracias a que en nuestro handle-error es un Componente inteligente o container el cual no retorna UI más bien retorna a los hijos que si contienen la UI.

regular-error:

```
import React from 'react';
function RegularError() {
  return (
    <h1 style={{color:'white'}}>Ohhh ha ocurrido un error lamentable :(</h1>
  )
}
export default RegularError;
```

Ahora si queremos que este componente no nuble toda la página, sino ser más especifico con el manejador de errores, pues es muy sencillo solo tenemos que utilizar el componente que acabamos de crear y envolver la section donde queremos manejar el error.

↑ volver al inicio

Creando reproductor de Video

Enlazando Eventos a React

Vamos a crear un reproductor de video usando los datos que tenemos de nuestra API, vamos a utilizar nuestro modal para contener al video, el cual se habre cada vez que le damos click a un elemento Media.

Elegimos construir un reproductor de video ya que un reproductor de video maneja muchísimos eventos dentro del DOM, muchísima interacción con el usuario.

Primero nuestro reproductor va tener un gran estado y cambios de los mismos, nuestro reproductor tiene un contenedor, el cual nuestro repdrocutor será un contenedor y necesitará un componente de UI para manejar su grapper, entonces hagamos esos dos para manjear está clase.

Vamos a crear un directorio especial para el repodructor el cual tendra sus componentes y containers:

src/player/containers/video-player.js

nuestro container video-player.js es nuestro contenedor del video el cual va a retornar a la UI de video-player.

```
import React from 'react';
import VideoPlayerLayout from '../components/video-player';
class VideoPlayer extends React.Component {
  render() {
    return (
      <VideoPlayerLayout>
        <video
          controls
          autoplay
```

```
            src=""
          >
        </VideoPlayerLayout>
      )
    }
  }
  export default VideoPlayer;
```

Layout del videoPlayer: **src/player/components/video-player.js** Este componente va a renderizar lo que tengan sus hijos lo que vengan por propiedades del children.

```
import React from 'react';
const VideoPlayerLayout = (props) => (
  <div  className="VideoPlayer">
    {props.children}
  </div>
)
export default VidePlayerLayout;
```

Nuestro VideoPlayer tiene que estar en nuestra HOME, porque nuestra home es nuestro Container de la página principal, es por eso que aquí importaremos el video. `ìmport VideoPlayer from '../../player/containers/video-player.jsx';` Aunque tiene que estár dentro del Modal lo vamos a poner tirado en el body para ir modificando sus propiedes con detalle.

↑ volver al inicio

Componentes de Título y Fuente de Vídeo

Ya iniciamos con nuestro reproductor de video y vamos a ir paso a paso incluyendole nuevos feachures a esté reproductor de video. Lo que vamos a hacer ahora es poner dentro de un Componente esté pedacito donde tenemos un video, porque estó de acá va a intervenir muchísimo lo cual iremos viendo en futuras clases. Otra cosa que vamos a hacer es crear un componente muy sencillo que nos muestre el titulo del video, el nombre del video que estamos reproduciendo.

Componente de video: src/player/components/video.js

```
import React from 'react';
import './video.css';
class Video extends React.Component {
  render() {
    return (
      <div className="Video">
        <video
          autoPlay={this.props.autoplay}
          src={this.props.src}
        />
      </div>
```

```
    )
  }
}
export default Video;
```

¿Tal vez te preguntarás? ¿Porque hemos hecho un Componente con estado(inteligente) y lo puso dentro de un componente que es de IU?

Es sencillo porque una de las reglas que teníamos era evitar al máximo que nuestros componente que teníamos que solo eran de UI vallan a estar teniendo lógica, en esté caso no aplica mucho nuestro push principal, porque este componente de video tiene métodos, así como {VideoPlay, VideoPause, VideoStop} y también tiene muchos eventos que se van a ir lanzando como por ejemplo: eventos de cambio de tiempo, de si lo estamos pausando, si estamos haciendo algó diferente con el video, así que hay que hacer algo muy interactivo y por eso lo estamos encapsulando dentro de otro lado para exonerar esas responsabilidades y manejarlas de una manera mucho más de visualizar.

video-player modificado:

```
import React from 'react';
import VideoPlayerLayout from '../components/video-player';
import Video from '../components/video';
class VideoPlayer extends React.Component {
  render() {
    return (
      <VideoPlayerLayout>
        <Video
          autoplay={true}
          src="ruta...."
        />
      </VideoPlayerLayout>
    )
  }
}
export default VideoPlayer;
```

Ya tenemos el Video externalizado, otra cosa que vamos a externalizar es el nombre del video que estamos reproduciendo, el cual será un componente que llamaremos `<Title />` componente title:
src/player/components/title.js

```
import React from 'react';
import './title.css';
const Title = (props) => (
  <div className="Title">
    <h2>{props.title}</h2>
  </div>
);
export default Title
```

Video es el componente que consumirá este componente y quedaría así.

```
import React from 'react';
import './video.css';
import Title from './title';
class Video extends React.Component {
  render() {
    return (
      <div className="Video">
        <Title title={this.props.title}/>
        <video
          autoPlay={this.props.autoplay}
          src={this.props.src}
        />
      </div>
    )
  }
}
export default Video;
```

↑ volver al inicio

Play / Pausa: Creando la UI

Continuando con nuestro reproductor de video continuaremos con las acciones, como reproductor es hecho desde 0, nosotros tenemos que configurarle los controles y para ello ocuparemos muchos eventos para el video, empezaremos con Play y Pause. Primero tenemos que hacer la UI y luego tenemos que hacer las acciones para interactuar con nuestro componente video.

Vamos a importar nuestro componente Play-Pause dentro del contenedor 'video-player.js' y vamos a crearlo. video-player modificado:

```
import React from 'react';
import VideoPlayerLayout from '../components/video-player';
import Video from '../components/video';
import Title from '../components/title';
import PlayPause from '../components/play-pause';
class VideoPlayer extends React.Component {
  render() {
    return (
      <VideoPlayerLayout>
        <Title
          title="Esto es un video chido!!"
        />
        <PlayPause />
        <video
          autoplay
          src="video....url"
        />
      </VideoPlayerLayout>
```

```
    )
  }
}
export default VideoPlayer
```

Component play-pause: Modificado y Agregando bottones.

```
import React from 'react';
import Play from '../../icons/components/play';
import Play from '../../icons/components/pause';
import './play-pause.css';
function PlayPause(props) {
  return (
    <div className="PlayPause">
      <button onClick={props.handleClick}>
        <Play
          size={25} color="white"
        />
      </button>

      <button onClick={props.handleClick}>
        <Pause
          size={25} color="white"
        />
      </button>
    </div>
  )
}
export default PlayPause;
```

Las eventos de Click vienen como propiedades de el contenedor y componente inteligente, las cuales las crearemos ahí mismo y le daremos un estado inicial.

Componente VideoPlayer con estado y métodos que hereda PlayPause

```
import React from 'react';
import VideoPlayerLayout from '../components/video-player';
import Video from '../components/video';
import Title from '../components/title';
import PlayPause from '../components/play-pause';
class VideoPlayer extends React.Component {
  state = {
    pause: true,
  }
  togglePlay = (event) => {
    this.setState({
      pause: !this.pause
    })
  }
```

```
  render() {
    return (
      <VideoPlayerLayout>
        <Title
          title="Esto es un video chido!!"
        />
        <PlayPause
         // Vamos a agregarle la opcion de Pause para decirle que botón mostrar en
esté elemento.
          active={this.state.pause}
          handleClick={this.togglePlay}
        />
        <video
          autoplay
          src="video....url"
        >
      </VideoPlayerLayout>
    )
  }
}
export default VideoPlayer
```

Recibimos una propiedad adicional que nos dice si el video está pausado o no y de esté modos mostrar el botton correcto, para saber que botón mostras haremos una validación.

```
import React from 'react';
import Play from '../../icons/components/play';
import Play from '../../icons/components/pause';
import './play-pause.css';
function PlayPause(props) {
  return (
    <div className="PlayPause">
      {
        props.pause ?
        <button onClick={props.handleClick}>
          <Play
            size={25} color="white"
          />
        </button>
      :
        <button onClick={props.handleClick}>
          <Pause
            size={25} color="white"
          />
        </button>
      }
    </div>
  )
}
export default PlayPause;
```

↑ volver al inicio

Agregando lógica al botón de Play y Pausa

Nuestro botón de Pause esta en el Dom pero ahora nos falta que realice alguna opción en el reproductor de video.

Lo primero que tenemos que hacer es manejar esté estado inicial de esté boton de play-pause para que corresponda a las propiedades que le llegan a mi reproductor de video, en esté caso una reproducción automatica. Esto lo podemos hacer muy sencillo **utilizando el ciclo de vida** de el *VideoPlayer* y utilizando la parte cuando el componente ya se ha montado **componentDidMount** para validar algó especial; aquí podemos cambiar algo en el estado con referencia a las propiedades que llegan a nuestro VideoPlayer, a nuestro Componente nos autoplay como propiedad la cual podemos consumir como propiedad. Ahora que estó viene como propiedad podemos hacerle una referencia en el *componentDidMount* para verificar el estado en el que estamos y cambiar el estado lo cual lo haremos desde el Componente *'video-player.js'*

```
import React from 'react';
import VideoPlayerLayout from '../components/video-player';
import Video from '../components/video';
import Title from '../components/title';
import PlayPause from '../components/play-pause';
class VideoPlayer extends React.Component {
  state = {
    pause: true,
  }
  togglePlay = (event) => {
    this.setState({
      pause: !this.pause
    })
  }
  componentDidMount() {
    this.setState({
      pause: (!this.props.autoplay)
    })
  }
  render() {
    return (
      <VideoPlayerLayout>
        <Title
          title="Esto es un video chido!!"
        />
        <PlayPause
         // Vamos a agregarle la opcion de Pause para decirle que botón mostrar en
esté elemento.
          active={this.state.pause}
          handleClick={this.togglePlay}
        />
        <video
          autoplay={this.props.autoplay}
          src="video....url"
        >
```

```
        </VideoPlayerLayout>
      )
    }
  }
  export default VideoPlayer
```

Ahora manejaremos el evento de pausar el video usando el Icon de Pause.

- Lo primero que tenemos que hacer es que sea configurable esté play-pause que lo tenemos configurable por medio del reproductor de video, pero no en nuestro componente de video. Porque nuestro componente de video mandá el estado del mismo, si está pausado, en que minuto está y todas las demás propiedades del video.

Tenemos que pasar el video como propiedad para que puedamos manipular las propiedades dentro del componente de Video. Primero vamos a poner la propiedad de pause en video y le vamos a gregar el evento de pause. Ahora vamos a nuestro Componente Video donde ya estamos recibiendo una propiedad de si ya está pausado o no está pausado, pero como por defecto ya estamos utilizando autoplay para prender o apagar el video en el estado inicial.

Pero para los diferentes estados que va a tener, es decir para cuando le demos en pause cambiaría ese estado, **vamos a utilizar el ciclo de vida del video** para recibir esas propiedes y validar que hacer con eso para eso ocuparemos el componente **componentWillReceiveProps** que funciona para recibir nuevas propiedades, donde nos llegán las siguientes-propiedes.

Acá podemos validar las propiedades con las propiedades que me están llegando y las propiedades que el componente ya tiene, para ver si hay algún cambió y despues de eso ejecutar algó. Si las nuevas pripiedades son diferentes de las propiedes que ya están tenemos que hacer un toggle del play dentro del video. Para eso crearemos un función que maneje el evento toggle.

Lo curioso de estó es que necesitamos que pausar a que le damos Play y a que Pauso. Para hacer esto dentro de nuestro elemento de video tenemos que hacer una referencia a el mismo, porque esté elemento HTML es el que realmente se pausa o se prende, que sería una propiedad que tiene el elemento de video, para ello usaremos una referencia.

Con la referencia a nuestro video HTML ejecutamos una función setRef la cual vamos a crear, esta función obtiene el elemento del ref, ahora que tenemos el elemento y lo recibimos en la funcion podemos asigarle el evento a nuestro estado de video, de la siguiente manera:

```
setRef = element => {
    this.video = element
  }
```

Una vez recibido el elemento ya podemos usar esa propiedad de video el cual tiene todos los métodos de nuestro elemento de video. y ahora podemos usarlo en la función de togglePlay: En está función podemos validar que el elemeto se Pause o haga Play deacuerdo a las circunstancias de nuestro estado pause.

```

```

```
togglePlay() {
  if (this.props.pause) {
    this.video.play();
  } else {
    this.video.pause();
  }
}
```

```
import React from 'react';
import './video.css';
import Title from './title';
class Video extends React.Component {
  togglePlay() {
    if(this.props.pause) {
      this.video.play();
    } else {
      this.video.pause();
    }
  }
  componentWillReceiveProps(nextProps) {
    if(nextProps.pause !== this.props.pause) {
      this.togglePlay();
    }
  }
  setRef = element => {
    this.video = element
  }
  render() {
    return (
      <div className="Video">
        <Title title={this.props.title}/>
        <video
          autoPlay={this.props.autoplay}
          src={this.props.src}
          ref={this.props.setRef}
        />
      </div>
    )
  }
}
export default Video;
```

↑ volver al inicio

Duración de Video

Los datos de la duración del video que tinen normalmente todos los reproductores, estos datos no los tenemos en una propiedad, no los podemos calcular, el media de esté video, lo tiene dentro de sus

metadatos, así que tenemos que detectar un par de elementos que tiene nuestro componente Video en su tag `<video />`Y son los Media events que maneja React, podemos consultarlo todos en MediaEvents. Pero el MediaEvent que necesitamos ahora es el **onLoadedMetadata** porque en la metadata es donde vamos a tener la duración del video. Por ejemplo puede tener el nombre metido dentro de sus metaDatos, pero lo que si seguramente tiene es la duración del video.

vamos a utitilizar onLadadedMetadata como key dentro del tag de video, este evento lo vamos a recibir como propiedad o como función del mismo componente, en esté caso lo haremos desde su padre 'video-player'.

Algo adicional que podemos hacer al momento de recibir propiedades es destructurar el objeto de propiedades y crear valores dentro de un objeto, a esté método se le llama destructur, y sirve para destructurar el objeto por medio de sus keys. ejemplo:

```
const = {
  handleClick
  handleChange
  event
  event2
  etc..
} = this.props
```

```
import React from 'react';
import './video.css';
import Title from './title';
class Video extends React.Component {
  togglePlay() {
    if(this.props.pause) {
      this.video.play();
    } else {
      this.video.pause();
    }
  }
  componentWillReceiveProps(nextProps) {
    if(nextProps.pause !== this.props.pause) {
      this.togglePlay();
    }
  }
  setRef = element => {
    this.video = element
  }
  render() {
    const  {
      handleLoadedMetadata
    } = this.props
    return (
      <div className="Video">
        <Title title={this.props.title}/>
        <video
          autoPlay={this.props.autoplay}
```

```
            src={this.props.src}
            ref={this.props.setRef}
            onLoadedMetadata={handleLoadedMetadata}
          />
        </div>
      )
    }
  }
  export default Video;
```

Ahora como la propiedad de *handleLoadedMetadata* nos llegá desde el padre, es decir de 'video-player' tenemos que enviarle la propiedad a el componente de Video y crear la función que envía esta propiedad.

Enviando **handleLoadedMetadata** como propiedad y creando la función en 'video-player':

```
import React from 'react';
import VideoPlayerLayout from '../components/video-player';
import Video from '../components/video';
import Title from '../components/title';
import PlayPause from '../components/play-pause';
class VideoPlayer extends React.Component {
  state = {
    pause: true,
    duration: 0,
  }
  togglePlay = (event) => {
    this.setState({
      pause: !this.pause
    })
  }
  componentDidMount() {
    this.setState({
      pause: (!this.props.autoplay)
    })
  }
  handleLoadedMetadata = event => {
    this.video = event.target;
    this.video.duration
    this.setState({
      duration: this.video.duration
    })
  }
  render() {
    return (
      <VideoPlayerLayout>
        <Title
          title="Esto es un video chido!!"
        />
        <PlayPause
         // Vamos a agregarle la opcion de Pause para decirle que botón mostrar en
esté elemento.
```

```
            active={this.state.pause}
            handleClick={this.togglePlay}
          />
          <video
            autoplay={this.props.autoplay}
            src="video....url"
            handleLoadedMetadata={this.handleLoadedMetadata}
          />
        </VideoPlayerLayout>
      )
    }
  }
  export default VideoPlayer
```

En nuestra función de **handleLoadedMetadata:** Primero vamos a tener dentro del evento quién disparó esté evento que sería 'event.target', este elemento lo vamos a igualar a una propiedad para despues poderlo manipular tal como lo hicimos con el componente de video, que sería: `this.video = event.target`

Lo que necesitamos saber es otro dato del evento de la cargada de la metadata para saber cuanto dura esté video, y es una propiedad que tiene el elemento de video que acabamos de crear y lo obtiene apartir de que se cargue su metadata que es la duración, la cual vamos a asignarsela a el estado.

```
    state = {
      pause: true,
      duration: 0,
    }
  handleLoadedMetadata = event => {
    this.video = event.target
    setState({
      duration: this.video.duration
    })
  }
```

Ahora tenemos que tener un lugar para mostrar la duración de esté video que sería un nuevo componente del Timer, el cual crearemos a continuación y agregaremos a nuestro mismo 'video-player'

Timer: **src/player/components/player.jsx**

```
import React from 'react';
import './timer.css'
Timer = (props) => (
  <div className="Timer">
    <p>
      <span>00 / {props.duration}</span>
    </p>
  </div>
)
export default Timer;
```

Para no olvidemos de enviarle las propiedades conrrespondientes:

```
import Timer from '../components/timer.jsx';
<Timer
  duration={this.state.duration}
/>
```

Ahora vamos a crear un Componente para los Controles y poder tener los controles en el lugar correcto. Esté componente solo va a retornar a sus hijos en este caso los controles. src/player/components/video-player-controls.js

```
import React from 'react';
import './video-player-controls.css';
function VideoPlayerControls(props) {
  return (
    <div className="VideoPlayerControl">
      {props.children}
    </div>
  )
}
export default VideoPlayerControls
```

Ahora que hemos creado nuestro componente pongamoslo dentro del video-player envolviendo a sus hijos.

```
import React from 'react';
import VideoPlayerLayout from '../components/video-player';
import Video from '../components/video';
import Title from '../components/title';
import PlayPause from '../components/play-pause';
import Timer from '../components/timer.jsx';
import Controls from '../components/video-player-controls.jsx'
class VideoPlayer extends React.Component {
  state = {
    pause: true,
    duration: 0,
  }
  togglePlay = (event) => {
    this.setState({
      pause: !this.pause
    })
  }
  componentDidMount() {
    this.setState({
      pause: (!this.props.autoplay)
    })
  }
  handleLoadedMetadata = event => {
    this.video = event.target;
```

```
    this.video.duration
    this.setState({
      duration: this.video.duration
    })
  }
  render() {
    return (
      <VideoPlayerLayout>
        <Title
          title="Esto es un video chido!!"
        />
        <Controls>
          <PlayPause
          // Vamos a agregarle la opcion de Pause para decirle que botón mostrar
en esté elemento.
            active={this.state.pause}
            handleClick={this.togglePlay}
          />
          <Timer
            duration={this.state.duration}
          />
        </Controls>
        <video
          autoplay={this.props.autoplay}
          src="video....url"
          handleLoadedMetadata={this.handleLoadedMetadata}
        />
      </VideoPlayerLayout>
    )
  }
}
export default VideoPlayer
```

↑ volver al inicio

Tiempo transcurrido

Ya sabemos cuanto dura nuestro video gracias a la metadata pero no sabemos en que posición del video estamos en esté momento, eso es lo que vamos a resolver en esta clase, el tiempo transcurrido y para eso tenemos **onTimeUpdate** el cual es un evento del video, el cual vamos a consumir desde el Contenedor Padre, crearemos un método para recibir evento de **onTimeUpdate** para despues manipularlo. consumiento **onTimeUpdate** desde video:

```
return (
  <div>
    <video
      autoPlay={this.props.autoplay}
      src={this.props.src}
      ref={this.setRef}
      onLoadedMetadata={handleLoadedMetadata}
      onTimeUpdate={hanldeTimeUpdate}
```

```
      />
    </div>
  )
```

Evento **hanldeTimeUpdate** que vamos a enviar desde 'video-player':

```
  state = {
    ..,
    currentTime: 0
  }
handleTimeUpdate = event => {
  console.log(this.video.currentTime)
  this.setState({
    currentTime: this.video.currentTime
  })
}
<Video
  autoPlay={this.props.autoplay}
  src={this.state.src}
  handleLoadedMetadata={this.handleLoadedMetadata}
  handleTimeUpdate={this.handleTimeUpdate}
/>
```

Lo que hacemos en método **hanldeTimeUpdate** es imprimir en cosola el currentTime que es el tiempo que está transcurriendo del video. Este dato lo enviamos a un estado y esté estado lo imprimimos en el Dom para que de esta manera puedamos ver el tiempo que está transcurriendo del video.

Aunque estamos imprimiendo los datos del video como el timepo y tiempo transcurrido este tiempo se está mostrando en segundos con milisegundos, lo cual nosotros no queremos que se vea, para esto lo que tenemos que hacer es darle un formato de segundos y minutos a nuestro contador y para hacer esto usemos un poquitín de JavaScript

Vamos a crear un par de funciones en un archivo que se encuentra en: **src/player/utilities/format-time.js**

en esté archivo escribiremos 2 funciones, una para el formato segundos y minutos y otra para que el contador de segundos tenga base '00' y no solo un número entero.

```
function leftPad(number) {
   const pad = '00';
   return pad.substring(0, pad.length - number.length) + number;
}

function formattedTime(secs) {
 const minute = parseInt(secs / 60, 10)
 const seconds = parseInt(secs % 60, 10)
 return `${minute} : ${leftPad(seconds.toString())}`
}
```

```
export default formattedTime;
```

Está función la vamos a ocupar para guardar el currentTime en el formato adecuado y así imprimir los datos de manera correcta.

Para ello vamos a exportar la función que da el formato y la vamos a importar en nuestro 'video-player'.

```
import formattedTime from '../utilities/format-time';
state = {
    pause: true,
    duration: 0,
    totalDuracion: 0,
    currentTime: 0,
    value: 0,
    time: 0,
    loading: false,
    volume: 1,
}
handleLoadedMetadata = event => {
    this.video = event.target;
    this.setState({
      totalDuracion: this.video.duration,
      duration: formattedTime(this.video.duration),
    })
  }
  handleTimeUpdated = event => {
    // console.log(this.video.currentTime)
    this.setState({
      currentTime: formattedTime(this.video.currentTime),
      time: this.video.currentTime
    })
  }

  <Timer
    duration={this.state.duration}
    currentTime={this.state.currentTime}
  />
```

↑ volver al inicio

Barra de Progreso

Ahora vamos a crear la barra para desplazar el video es decir poder movernos del minuto 5 al minuto 1, etc. La cual sería nuestras barra de progreso, nuestra ProgressBar la cual también será un componente Componente 'progress-bar.jsx';

```
import react from 'react';
import './progress-bar.css'
```

```
const = ProgressBar(props) => (
  <div className="ProgressBar">
    <input
      type="range"
      min={0}
      max={props.duration}
      value={props.value}
    />
  </div>
)
export default ProgressBar;
```

Esté componente va a recibir el rango máximo que recibirá el elemento de rango. Ahora vamos a manejar los evento que haiga de actualización de tiempo para darle el valor actual a esto, como ejemplo nosotros podemos settear el valor al input con la propiedad de *value* el cúal debe ser dinámico a como va avanzando esté tiempo, para ello vamos a enviarle el valor de la propiedad value desde el contenedor de este componente donde vamos a igual el value a la propiedad dinámica de nuestro **currentTime**. en el archivo 'video-player' ejemplo:

```
import ProgressBar from '../components/progress-bar.jsx';
<ProgressBar
  duration={this.state.duration}
  value={this.state.currentTime}
/>
```

Ahora ya que estamos settean el valor de value, ya no podemos mover las barra de progreso eso es porque ya estamos manejando el valor de ese elemento, así que deberíamos manejar los cambios de la barra de progreso para eso usaremos el envento **'onChange'** que sería igual a una propiedad que esperamos recibir de el 'video-player' el cuál será el manejador de estos cambios.

manejador de cambios en ProgressBar:

```
import react from 'react';
import './progress-bar.css'
const = ProgressBar(props) => (
  <div className="ProgressBar">
    <input
      type="range"
      min={0}
      max={props.duration}
      value={props.value}
      onChange={props.handleProgressChange}
    />
  </div>
)
export default ProgressBar;
```

En el contenedor 'video-player.js' :

```
import ProgressBar from '../components/progress-bar.jsx';
handleProgressChange = envent => {
  this.video.currentTime = event.target.value
}
<ProgressBar
  duration={this.state.duration}
  value={this.state.currentTime}
  handleProgressChange={this.handleProgressChange}
/>
```

En el ejemplo anterior estamos creando la función que maneja los cambios del input en donde estamos setteando el tiempo del video al tiempo que especifica la barra de progresso.

↑ volver al inicio

Spinner

Ahora que podemos movernos a lo largo del video, pero hay un momento en el que el video tiene que cargar datos y lo que hacen los reproductores cuando esto ocurre es mostrar un spinner o elemento que indica que se están cargando datos.

Para eso vamos a crear nuestro componente Spinner: src/player/components/spinner.jsx

```
import React from 'react';
import './spinner.css'
function Spinner(props) {
  return (
    <div className="Spinner">
      <span>Cargando...</span>
    </div>
  )
}
export default Spinner
```

Ahora vamos a ir a nuestro Reproductor y vamos a cargar el Spinner para que el video se pueda mostrar en en video.

```
import Spinner from '../components/spinner.jsx';
<Spinner />
```

Ahora tenemos que validar que salga esté Spinner si y solo sí esta el elemento cargando y para eso podemos detectar ese evento dentro del video.

Tenemos 2 eventos:

- Uno es cuando empieza ese periodo de carga
- Dos es cuando ya se ha cargado todos los datos.

Son dos eventos 1 para ponerlo y otro para quitarlo. Esos son los evento naturales del **video**. Los eventos son: **onSeeking** y **onSeeked**

Eventos en el Component de Video

```
return (
  <div>
    <video
      autoPlay={this.props.autoplay}
      src={this.props.src}
      ref={this.setRef}
      onLoadedMetadata={handleLoadedMetadata}
      onTimeUpdate={hanldeTimeUpdate}
      onSeeking={props.handleSeeking}
      onSeeked={props.handleSeeked}
    />
  </div>
)
```

Estás propiedades las vamos a enviar desde nuestro 'video-player.js'

```
state = {
  ..: ..,
  etc: ..,
  loading: false,
}
handleSeeking = event => {
  // Cargando Video
  this.setState({
    loading: true
  })
}
handleSeeked = event => {
  // Video Cargado
  this.setState({
    loading: false
  })
}
<Video
  autoPlay={this.props.autoplay}
  src={this.state.src}
  handleLoadedMetadata={this.handleLoadedMetadata}
  handleTimeUpdate={this.handleTimeUpdate}
  handleSeeking={this.handleSeeking}
  handleSeeked={this.handleSeeked}

/>
```

Con estos 2 eventos vamos a manejar un nuevo estado que se llamá loading: esté estado pasará de true a false cuando el video este cargando y deje de cargar, este estado activará nuestro loader.

Nuestro VideoPlayer ya sabe que estó ocurre pero el Spinner aún no, para ello vamos a crear una validación usando el loadig dentro del spinner.

en VideoPlayer enviamos Active para consumirla dentro de Spinner

```
import Spinner from '../components/spinner.jsx';
<Spinner
  active={this.state.loading}
/>
```

Consumiendo active y validando Spinner;

```
import React from 'react';
import './spinner.css'
function Spinner(props) {
  if(props.active) return null;
  return (
    <div className="Spinner">
      <span>Cargando...</span>
    </div>
  )
}
export default Spinner
```

↑ volver al inicio

Control de Volumen

En esta parte vamos a controlar el volumen del video. Primerov vamos a crear un nuevo Componente que se va a llamar Volumen el cual deberá de renderizar en los controles del video.

Algo que tenemos que saber es que el Volumen dentro de los elementos de Media de html5 funciona con valores entre 0 y 1, lo que quiere decir es que 0 es no se escucha nada y 1 es que se escucha a full volumen.

```
import React from 'react';
import VolumeIcon from '../../icons/components/volume';
import './volume.css';
const Volume = (props) => (
  <button
    onClick={props.handleClick}
    className="Volume"
  >
    <VolumeIcon
```

```
        color="white"
        size={25}
      />
      <div className="Volume-range">
        <input
          type="range"
          min={0}
          max={1}
          step={0.05}
          onChange={props.handleChange}
          defaultValue={props.value}
        />
      </div>
    </button>
)
export defualt Volume
```

Ahora ya podemos exportar nuestro Volumen dentro de 'video-player';

```
import Volume from '../componets/volumen.jsx';
handleVolumeChange = event => {
    this.video.volume = event.target.value;
  }

handleMuteVolume = event => {
  this.video.muted = !this.video.muted;
  this.video.muted ?
    this.setState({volume: 0})
  : this.setState({volume: 1});
}
<Volume
  handleChange={this.handleVolumeChange}
  handleClick={this.handleMuteVolume}
  value={this.state.volume}
/>
```

↑ volver al inicio

FullScreen

Estamos cerca de terminar el reproductor de video, de terminar el proyecto y tener una apliación completa hecha en React.js

Ahora crearemos nuestro Componente que tendra el botón de FullScreen el cual nos llevará a fullscreen o nos cerrará el fullscreen.

```
import React from 'react';
import FullScreenIcon from '../../icons/components/full-screen.js'
import './full-screen.css';
```

```
const FullScreen = () => (
  <div className="FullScreen">
    <FullScreenIcon
      color="white"
      size={25}
    />
  </div>
)
export default FullScreen;
```

Ahora la importaremos a nuestro Container 'video-player' y agregaremos el elemento a nuestros controles.

```
import FullScreen from '../components/full-screen.js'
<FullScreen
  handleClick={this.handleFullScreen}
/>
```

Ahora manejaremos el evento del botón que es el evento click la cúal nos llega como propiedad de 'video-player'.

Dentro de esté evento manejaremos el fullscreen, esto lo haremos con una API del navegador que nos ayuda a ponernos en FullScreen. Este ejemplo es hecho para Chrome ya que está api es deiferente para algunos navegadores.

Para eso Tenemos que saber que elemento deseamos mandar a FullScreen ¿Será el video? adicionalmente del video queremos todo el reproductor para poder manipular sus controles y para eso crearemos una referencia a Todo el Reproductor de video.

```
handleFullScreen = event => {
 if (!document.webkitIsFullScreen) {
   this.player.webkitRequestFullScreen()
 } else {
   document.webkitExitFullscreen()
 }
}
setRef = element => {
  this.player = element
}
<VideoPlayerLayout
  setRef={this.setRef}
>
  <Title>
  <Controls>...</Controls>
  <Spinner>
  <Video>
</VideoPlayerLayout>
```

Ahora iremos a VideoPlayerLayuout donde le llegará esta función de setRef.

```
import React from 'react';
import './video-player-layout.css';
const VideoPlayerLayout = (props) => (
  <div
    className="VideoPlayer"
    ref={props.setRef}
  >
    {props.children}
  </div>
)
export default VideoPlayerLayout;
```

En esté momento ya le hicimos la referencia al elemento que envuelve a todo nuestro reproductor y vamos a tener el elemento player disponible dentro de nuestro component video-player y podremos enviarlo a FullScreen.

↑ volver al inicio

FullScreen para todos los navegadores

Para este ejercicio seguimos la misma logica para poner nuestro elemento de this.player a fullscreen solo que aquí hacemos la validación para todos los navegadores.

1. Primero Creamos una Función Js para Saber desde que navegador estamos, está función nos retorna el nombre del navegador. Esta función la creamos como utilidades de nuestro proyecto. en src/player/utilities/navegator.js

```
function navegator() {
  const navegador = {
    webkit: navigator.userAgent.toLowerCase().indexOf('chrome') > -1,
    moz: navigator.userAgent.toLowerCase().indexOf('firefox') > -1,
    edge: navigator.userAgent.toLowerCase().indexOf('safari') > -1
  }
  if(navegador.webkit) return "chrome"
  else if(navegador.moz) return "moz"
  else if(navegador.edge) return "safari"
}
export default navegator;
```

Ahora está función la consumiremos en un **Case** para saber que navegador estamos ocupando y retornar las funciones necesarias para cada tipo de navegador de la siguiente manera.

EventoClick que envía el Componente a FullScreen

```
handleFullScreen = event => {
    switch(Navegador()) {
      case "chrome":
```

```
        !document.webkitIsFullScreen ?
          this.player.webkitRequestFullscreen()
        : document.webkitExitFullscreen();
        break
      case "moz":
        !this.player.mozFullScreen ?
          this.player.mozRequestFullScreen()
        : mozCancelFullScreen();
        break;
      default:
        !this.player.msFullscreenEnabled ?
          document.msRequestFullscreen()
        : document.     msExitFullscreen();
        break;
    }
  }
```

↑ volver al inicio

Puliendo detalles

Nuestro proyecto está completamente listo solo nos falta pulirle algúnos detalles y poner las cosas donde son, evniarle las propiedades correctas y agregarle los estilos que hagán falta.

1. Vamos a mover el componente Video dentro de Modal dentro del archivo contenedor 'Home'.

```
import React from 'react';
import HomeLayout from '../components/home-layout.jsx';
import Categories from '../../categories/components/categories.jsx';
import Related from '../components/related.jsx';
import ModalContainer from '../../widgets/container/modal.jsx';
import Modal from '../../widgets/components/modal.jsx';
import HandleError from '../../error/containers/handle-error.jsx';
import VideoPlayer from '../../player/containers/video-player.jsx';

class Home extends React.Component {
  state = {
    modalVisible: false,
  }
  handleOpenModal = media => {
    this.setState({
      modalVisible: true,
      media: media
    })
  }
  handleCloseModal = (event) => {
    this.setState({
      modalVisible: false,
    })
  }
```

```
  render() {
    return (
      <HandleError>
        <HomeLayout>
        <Related />
          <Categories
            categories={this.props.data.categories}
            handleOpenModal={this.handleOpenModal}
          />
        {
          this.state.modalVisible &&
          <ModalContainer>
            <Modal handleClick={this.handleCloseModal}>
              <VideoPlayer
                autoplay
                src={this.state.media.src}
                title={this.state.media.title}
              />
            </Modal>
          </ModalContainer>
        }
        </HomeLayout>
      </HandleError>
    )
  }
}

export default Home;
```

despues cambiaremos un poco los estilos de nuestro control:

```
.VideoPlayerControls {
  display: flex;
  position: absolute;
  background: rgba(0,0,0,.7);
  bottom: 10px;
  left: 10px;
  right: 10px;
  z-index: 2;
  border-radius: 20px;
  padding: 0 10px;
}
```

Vamos a agregarle estilos al botón que cierra el modal:

```
import React from 'react';
import './modal.css';
function Modal(props) {
  return (
```

```
      <div className="Modal">
        {props.children}
        <button
          onClick={props.handleClick}
          className="Modal-close"
        />
      </div>
    )
  }
  export default Modal;
```

Para el gran final queremos hacer que nuestro contenido del video, sea el contenido sea el contenido que está en cada uno de mis elementos que manda el API, y si bien cuando hacemos un click estamos abriendo el modal no le estamos enviando nigun dato adicional para terminar el proceso, así que hagamos esa parte, lo cual está dentro de nuestro Media.js

Media.js cuando le ejecuta su click manda a una propiedad que se llamá handleClick que le llegá aunque podemos hacer diferente, podemos manejar el click en el mismo componente de Media y ejecutar sí la propiedad handleClick que nos llegá pero enviarle si un párametro adicional.

aquí adentro vamos a llamar a nuestro **this.props.handleClick** pero adiconalmente le vamos a enviar las propiedades, ejemplo: **this.props.handleClick(this.props)**

Esto quiere decir que cuando le damos Click estamos ejecutando un método handleClick que a su vez ejecuta un método openModal

```
class Media extends React.Component {
  handleClick = (event) => {
      this.props.openModal(this.props)
  }
  render() {
      return (
        <div className="Media" onClick={this.handleClick}>}
        ...
      )
  }
}
```

handleClick está ejecutando openModal que viene como propiedad de su padre 'Playlist.jsx'

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

function Playlist(props) {
  return (
    <div className="Playlist">
        {
          props.playlist.map((item) => {
```

```
            return (
              <Media
                {...item}
                key={item.id}
                openModal={props.handleOpenModal}
              />
            )
          })
        }
      </div>
  )
}
export default Playlist;
```

Playlist envía la función openModal que a su vez viene desde el componente de Categories.

```
import React from 'react';
import Category from './category.jsx';
import './categories.css'
import Search from '../../widgets/container/search.jsx';
function Categories(props) {
  return (
    <div className="Categories">
    <Search />
      {
        props.categories.map((item) => {
          return (
            <Category
              key={item.id}
              {...item}
              handleOpenModal={props.handleOpenModal}
            />
          )
        })
      }
    </div>
  )
}
export default Categories;
```

Esta función le llega como propiedad también desde el Contenedor Padre Home el cual si está definiendo la función:

```
handleOpenModal = media => {
    this.setState({
      modalVisible: true,
      media: media
    })
}
```

```
<Categories
  categories={this.props.data.categories}
  handleOpenModal={this.handleOpenModal}
/>
```

Como vimos en Media.js está función de OpenModal ahora lleva propiedades que tiene nuestro Media la cual le pasamos como otra propiedad dentro de la misma función OpenModal

La función lleva esta dirección Home -> Categories -> Category -> Playlist -> Media {Aquí Media envia las propiedes a la función openModal}.

La función OpenModal recibe como parametro Todas las propiedades de Media y las guarda como un objeto Media dentro del estado del componente Home es decir como objeto 'State'

Ahora la propiedad Media es consumida por el VideoPlayer el cual ocupa esas propiedades para reproducir el video que se está ejecutando al darle click a Media.

```
// Creamos Media en el State con los nuevos valores
handleOpenModal = media => {
    this.setState({
      modalVisible: true,
      media: media
    })
  }
```

Rendereamos el VideoPlayer con las nuevas propiedades de Media.

```
<VideoPlayer
  autoplay
  src={this.state.media.src}
  title={this.state.media.title}
/>
```

Ahora que enviamos las propiedes dinámicas a VideoPlayer solo tenemos que consumirlas dentro del VideoPlayer que renderiza todos sus elementos.

```
<Title title={this.props.title}/>
<Video
  handleLoadedMetadata={this.handleLoadedMetadata}
  pause={this.state.pause}
  autoplay={this.props.autoplay}
  src={this.props.src}
  handleTimeUpdated={this.handleTimeUpdated}
  handleSeeking={this.handleSeeking}
  handleSeeked={this.handleSeeked}
/>
```

De esté modo obtenemos los datos dinámicos cada vez que le damos click a Media. Y si precionamos un click a cada Media diferente Obtenemos los nuevos datos del Media clickeado.

↑ volver al inicio

Producción

Una vez terminando el proyecto podemos compilar nuestro proyecto para obtener el proyecto de distribuición, para esto ejecutamos el script que creamos para compilar nuestro bundle que es un archivo de webpack para producción.

run `npm run build:prod`

Ya solo cambiamos las referencias de archivos a nuestro HTML y listo.

↑ volver al inicio