**¿Por qué aprender React?**

**React =** Librería de JavaScript para construir interfaces de usuarios.

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.

**Particularidades de ReactJS:**

* **Declarativo.** Es muy sencillo escribir interfaces y poderlas leer.
* **Basado en componentes.**
* **Aprende de una vez y escribe donde quieras.**

Requisitos:

Instalamos Node, también se instalara npm por default.

Instalamos un boilerplate (estructura de archivos y paquetes (o andamiaje) base para crear una aplicación.)

npm **install** -g **create**-react-app

Ahora para arrancar con el scafolding **create-react-app** platzi-video (te agrega webpack)

Luego hacemos una carpeta de webpack y colocamos el webpack-dev-config.js y el webpak.config.js

Porque vamos a tener uno de desarrollo donde el código corra super rápido y poder depurarlo y otra de producción donde el código corra super rápido pero en el navegador y que lo entiendan todos los navegadores.

Dentro de react iniciamos un proyecto de javascript con **npm init**

**Me va a crear el package.json**

**Coloco las dependencias en el package.json y luego en la consola npm i y ahí las instala.**

entry: {

"platzi-video": path.resolve(\_\_dirname, 'index.js'),

},

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'js/[name].js'

},

Cuando se compile lleva el index.js a una carpeta output que va a estar dentro de js/platzi-video.js

filename: filename: 'js/[name].[hash].js',

Si en el output, se coloca un hash, va a crear un javascript (js/platzi-video.8cafca12d414df804f12.js) con un nombre largo y cada vez que se compile, va a cambiar.

**Usando ReactDOM y JSX**

ReactDOM nos ayuda para renderizar en el navegador

ReactDOM.render recibe dos parámetros

1. Qué se va a renderizar.
2. Donde se va renderizar.

Instalaremos react y react-dom

--save las añade a nuestro package.json

npm install react react-dom --save-dev

**JSX**

Ahora veamos como se haría lo mismo pero empleando sintaxis JSX:

var Icon = (

<div className='icon-container'>

<img

src='icon-react.png'

className='icon-image'

/>

</div>)

ReactDOM.render(Icon, document.getElementById('app'))

**Estilos CSS**

**Estilos inline**

* Los estilos se crean como si fueran un objeto json.
* Por convensión, las propiedades se escriben en Cammel Case.
* El tamaño por defecto es px. Esto quiere decir que no es necesario agregarlo cuando esta sea la medida a aplicar.

**import** React, { Component } **from** "react";

**class** **Media** **extends** **Component** {

render() {

**const** styles = {

container: {

fontSize: 14

}

}

**return** (

<div style={styles.container}></div>

)

}

}

**export** **default** Media;

**Archivos .css**

Los archivos .css se importan con un import dentro del archivo .js o .jsx.

**import** "./media.css";

Para usarlos dentro de un elementos JSX se usa el atributos className.

<div className="Media">

Si quieren ver mi resumen completo lo pueden hacer [aquí](https://github.com/MineiToshio/CursosPlatzi/tree/master/Curso%20de%20React.js). Lo iré actualizado a medida que vaya avanzado con el curso.

**Validando tipado en propiedades**

npm install prop-types –save

<https://reactjs.org/docs/handling-events.html>

**Estado de los componentes en ReactJs**

as propiedades (props) son **inmutables**, es decir; estas no pueden cambiar. En el caso que quisiéramos que algo cambie en nuestro componente, hay que manejarlo con el **estado** de nuestros componentes, el cual permite tener contenido dinámico.

Para inicializar el estado tenemos que hacer uso de nuestro metodo **constructor**:

**constructor**(props) {

super(props);

this.state = {

author: props.author

}

}

Y en nuestros elementos vamos a utilizar **state** en lugar de **props**:

<p className="Media-author">{this.**state**.author}</p>

Para cambiar el estado tenemos un método especifico **setSate()** al cual le pasamos los valores que queremos modificar en nuestro estado.

**handleClick** = (event) => {

**this**.setState({

author: 'Emmanuel Alonso'

})

}

Con ES7 podemos hacer definir el estado directamente, en lugar de usar el constructor:

**state** = {

author: 'Ivan Robles'

}

Actualmente al crear un componente en react al inicializar los estdos se inicializa  
dentro del constructor como podemos ver en el siguiente ejemplo.

**import** React { Component } **from** "react";

**class** **MyComponent** **extends** **Component** {

consturctor(props) {

**super**(props);

// Inicializacion del los estados del componente.

**this**.state = {

nombre: "JMZ",

};

}

render() {

**const** { nombre } = **this**.state;

**const** { edad } = **this**.props;

**return** <h1>{`Nombre => ${nombre} ${edad}`}<h1>;

}

}

Como se puede ver en el ejemplo anterios estamos utilzando la forma basica para inicializar los estados de un componente, pero con los nuevos features de es2017 la estrucutra anterios podria cambiar a la siguiente manera.

**import** React { Component } **from** "react";

**class** **MyComponent** **extends** **Component** {

// Inicializacion de los estados.

state = {

nombre: "JMZ",

};

render() {

**const** { nombre } = **this**.state;

**const** { edad } = **this**.props;

**return** <h1>{`Nombre => ${nombre} ${edad}`}<h1>;

}

}

Como se muestra en el ejemplo anterior estamos quitando el **constructor** ya no es necesario para inicializar los estados del componente, es importante recalcar que con las nuevas verciones de **React** podemos omitir el uso de **super** para pasar los props al contructor de la clase **component**.

<h1>Estado de componentes</h1>

Las propiedades son inmutables. Estas no pueden cambiar. Si se desea hacer un cambio se usan los states.

## Creación del estado

**ES6**

En ES6 se tiene que construir un constructor y crear un state en el cual se le asignen las propiedades que se desea modificar.

**constructor**(props) {

**super**(props);

**this**.state = {

author: props.author

}

}

**ES7**

En ES7 solo se declara el state y se asignan las valores que se deseen usar.

state = {

author: "Sergio Minei"

}

## Modificar un estado

Para modificar un estado se usa la función setState.

**this**.setState({

author: "Ricardo Celis"

})

## Mostrar el valor del estado

Para mostrar el valor de un elemento de estado se usa this.stateseguido por el elemento que se desea usar.

<p className="Media-author">{this.state.author}</p>

**Ciclo de vida de los componentes**

**class** **MiComponente** **extends** **Components**{

**constructor**(){

// Enlazo (bind) eventos y/o inicializo estado

}

**componentWillMount**(){

// Se ejecuta antes de montar el componente

// Se podría usar para hacer un setState()

}

**render**(){

// Contiene todos los elementos a renderizar

// podrías usarlo para calcular propiedades (ej: concatenar una cadena)

}

**componentDidMount**(){

//Solo se lanza una vez

//Ideal para llamar a una API, hacer un setInteval, etc

}

//Actualización:

**componentWillReceiveProps**(){

//Es llamado cuando el componente recibe nuevas propiedades.

}

**shouldComponentUpdate**(){

//Idea para poner una condición y si las propiedades que le llegaron anteriormente

// eran las mismas que tenia retornar false para evitar re-renderear el componente

}

**componentWillUpdate**(){

//metodo llamado antes de re-renderizar el componente si shouldComponentUpdate

//devolvió true

}

// re-render si es necesario...

**componentDidUpdate**(){

//Método llamado luego del re-render

}

**componentWillUnmount**(){

//Método llamado antes de desmontar el componente

}

**componentDidCatch**(){

// Si ocurre algún error, lo capturo desde acá:

}

}

7

Ciclo de vida de los componentes de React:

* **Montado**: El momento en que el componente entra en escena.
* **Constructor**: método llamado antes de que el componente sea montado (componente aun no se ve).
  + Podemos iniciar el estado
  + Enlazar eventos (bind).
  + Es el primer método que se llama al instanciar un componente.
* **componentWillMount**: método llamado inmediatamente antes de que el componente se vaya a montar (componente aun no se ve).
  + Podemos hacer un setState() //Si tenemos alguna propiedad y queremos setear algún estado en base a ella
  + No hacer llamados a un API o suscripción a eventos. //Porque aun no tienes el dom
* **Render**: método que contiene todos los elementos a renderizar (estructura del componente).
  + Contiene JSX en el return.
  + Puedes calcular propiedades nCompleto = name + lastName.
* **componentDidMount**: Método llamado luego de montarse el componente (el componente ya está en la pantalla).
  + Solo se lanza una vez.
  + Enlazar (bind) de eventos.
  + Es el primer método que se llama al instanciar un componente.
  + Aquí podemos utilizar APIs (Navegador o Datos Externos).
* **Actualización**
* **componentWillReceiveProps**: método llamado al recibir nuevas propiedades que sirve para actualizar el estado con base a las nuevas propiedades.
* **shouldComponentUpdate**: método que condiciona si el componente se debe volver a renderizar, es utilizado para optimizar el rendimiento.
* **componentWillUpdate**: método llamado antes de re-renderizar un componente, es utilizado para optimizar el rendimiento.
* **Render**: método que realiza el re-render.
* **componentDidUpdate**: método llamado luego del re-render.
* **Desmontado**
* **componentWillUnmount**: método llamado antes de que el componente sea retirado de la aplicación.
* Manejo de Errores
* **componentDidCatch**: método llamado cuando ocurre un error al renderizar el componente, el manejo de errores solamente ocurre en componentes hijos. Podemos manejar errores del componente, dentro del componente, para que no se rompa toda la pantalla.

**Smart Components y Dumb Components**

**Componentes puros y funcionales en ReactJS**

**PureComponent**: tiene el método **shouldComponentUpdate** ya asignado (por defecto), si a este componente no se le actualizan las propiedades, no tenemos que validar a mano con shouldComponentUpdate, PureComponent lo hace por nosotros, es decir; si recibe nuevas propiedades pero son las que ya teniamos, no se re-renderiza.

**import** **React**, { **PureComponent** } from 'react';

**class** **Playlist** **extends** **PureComponent** {

render() {

<**Componente** />

}

}

**Componente Funcional**: Es una función la cual solo retorna el JSX de nuestro componente (renderiza UI), es más sencillo, más fácil de probar y este componente no tiene ciclo de vida.

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

**function** **Playlist**(props) {

**return** <Componente title={props.title} />

}

**Smart Components y Dumb Components**

Que hace vs Como se ve

Smart Dumb

Statefull Pure

Fat Skinny

Container Presentaional Component

**Presentacional** Cómo se ve

* Puede contener smart components u otros componentes de UI
* Permiten composición con `[props.children]``
* No depeden del resto de la aplicación
* No especifica cómo los datos son cargados o mutados

(Lo que recibe lo trata de imprimir directamente, solo muestra)

* Recibe datos y callbacks solo con propiedades

(Si en un componente funcional quiero manejar un clic, puedo poner una función encima de mi función componente, pero esto no se debería hacer asi, sino que la definición de la función debe venir como propiedad de un componente inteligente, el componente tonto solo lo v.)

* Rara vez tienen su propio estado
* Están escritos como componentes funcionales a menos que necesiten mejoras de performance. Sólo pueden ser **Componentes funcionales** o **Pure Components**

(Los componentes dumb pueden ser componentes funcionales o pure components, en el 99% son componentes funcionales)

**Containers** Qué hace

* Concetrado en el funcionamiento de la aplicación

(Tiene un ciclo de vida, se engarga del negocio, es como un manager le dice a un componente dumb bueno vos renderiza esto, vos lo otro, sabe lo que tiene la aplicación.)

* Contienen componentes de UI u otros containers
* No tienen estilos
* Proveen de datos a componentes de UI u otros contenedores
* Proveen de callbacks a la UI
* Normalmente tienen estado
* Llaman acciones
* Generados por higher order components

Separación de Funcionalidades (Como se ve, como funciona)

Un Componente tonto seria solo la vista del MVC

Un Componente inteligente seria el controlador del MVC

Mejora la capacidad de reutilizar componentes

Por ejemplo, si tenemos un botón con un cierto diseño, y este botón no maneja su propio clic entonces yo puedo usar ese botón en otro lado de la aplicación y al inyectarle por propiedad o parámetro la funcionalidad puedo hacer que ese botón haga otra cosa.

**Portales**

**Portales** es la manera en la que podemos renderizar componentes fuera del contenedor principal de la aplicación.

<**div** id="App"></**div**>

El caso de uso mas común son las ventanas modal.

Para crear un portal, se debe importar el metodo createPortal de react-dom:

**import** { createPortal} **from** 'react-dom';

El metodo createPortal() recibe dos parametros, al igual que con render es Lo que se va a renderizar y donde se va a renderizar:

**class** **ModalContainer** **extends** **Component** {

render() {

**return** (

createPortal(<**Component** />, document.getElementById('component-container'))

)

}

}

2