**M2S13: Fundamentos de React**

**Hola muy buen día, ¿cómo estás?, ya nos encontramos en la semana 13!!!, estoy muy feliz sé que has aprendido mucho y seguirás aprendiendo para que cada día seas mejor.**

**Antes de iniciar con el contenido me gustaría dejarte un video para que mejores tu respiración y esto te permita que puedas estudiar un poco más relajad@, bajes la ansiedad y el estrés, puedes ver el video**[**presionando aquí**](https://www.youtube.com/watch?v=I5tip6L5fOQ)

**¿Te sientes mejor?, sigamos adelante...**

**ÍNDICE**

* [React](https://github.com/U-Camp/BOOT-M2-SEM13#react)
* [Props](https://github.com/U-Camp/BOOT-M2-SEM13#props)
* [State](https://github.com/U-Camp/BOOT-M2-SEM13#state)
* [Ciclos de vida](https://github.com/U-Camp/BOOT-M2-SEM13#ciclos-de-vida)
* [Eventos](https://github.com/U-Camp/BOOT-M2-SEM13#eventos)
* [Error Boundaries](https://github.com/U-Camp/BOOT-M2-SEM13#error-boundaries)

**React**

Se ha hablado de manera general respecto al frontend y las tecnologías principales (HTML, CSS y Javascript) las cuales se usan para construir las vistas que el usuario va a observar y con las que va a interactuar directamente. React es precisamente una librería hecha y que usa Javascript que sirve para construir interfaces de usuario, conjuga y simplifica el uso de esas 3 tecnologías mediante el uso de componentes que encapsulan su propia lógica y mantienen un estado independiente que no afecta cualquier otro componente, básicamente es una librería de Javascript que fomenta el uso de componentes y es declarativo en cuanto al flujo de los datos, con esto quiere decir que siempre se pueden visualizar los cambios de estado del componente y en qué lugar dentro del layout se va a presentar dicho dato; es de código abierto. La librería fue creada por Facebook y sigue siendo mantenida por esta compañía.

La forma en la que React trabaja está basada en un Virtual DOM el cual es una representación en memoria del estado del DOM, lo que permite determinar que partes del DOM han cambiado de manera rápida y actualizando y renderizando únicamente aquellos componentes que hayan cambiado. A este proceso se le conoce como reconciliación, en las primeras versiones de React, este proceso estaba orquestado por el algoritmo Stack Reconciler, en versiones nuevas se implementó un nuevo algoritmo de reconciliación llamado Fiber.

Esta librería intenta estandarizar el concepto de componente y hace uso de JSX para definirlos. Un componente es una entidad que define un objeto compuesto por su propia funcionalidad y el diseño de este; los componentes permiten separar la interfaz de usuario en piezas independientes, reutilizables y obligan a pensar cada pieza en forma aislada.

Los componentes son precisamente pequeñas partes de una vista que realizan una actividad específica, y que pueden a su vez, componerse de otros componentes (Composición), lo que permite ir construyendo interfaces desde componentes que representan lo más sencillo (un botón, un label, un input) hasta componentes más complejos que se componen de estos componentes (un carrusel, una lista de artículos, una página web completa).

Un componente puede tener diferentes atributos que se usan para definir su estado, como lo son las props (propiedades) que son los datos iniciales del componente y el state que es la representación de los datos de un componente en un tiempo determinado, estos atributos se verán a profundidad en clases posteriores. Así mismo, un componente tiene métodos predefinidos que ayudan a ejecutar ciertas acciones en un momento determinado (por ejemplo cuando se requiere actualizar el estado o justo después de que el componente se monte en el Virtual DOM o cuando el componente se va a quitar de la vista del usuario), a éstos métodos se les conoce como Lifecycle Methods, sin embargo, en últimas versiones de ReactJS, los métodos del ciclo de vida y del manejo del estado se han ido deprecando o en su defecto han sido reemplazados por los hooks, los cuales son métodos específicos de cada componente que intentan resolver problemas de lógica y de entendimiento humano / máquina que los componentes de clase generan, el hook que reemplaza los métodos del ciclo de vida es el useEffect(), el cual desencadena un side effect y dependiendo de un arreglo de dependencias que recibe como parámetros, éste hook se ejecuta en determinada parte del ciclo de vida. Para el manejo de estado tenemos el useState(), el cual devuelve un arreglo con una variable de estado en la primera posición y un método para actualizar esa variable en la segunda posición.

Todas y cada una de estas formas de escribir componentes de react se verán a profundidad durante el transcurso del bootcamp.

<!-- Cargar React. -->

<script src="https://unpkg.com/react@16/umd/react.production.min.js" crossorigin></script>

<script src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js" crossorigin></script>

<!-- Cargar JSX. -->

<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>

<!-- React va a buscar en este elemento y montará el componente aquí -->

<div id="ejemplo-componente"></div>

<!-- Cargamos nuestro componente de React. -->

<script type="text/babel">

// Ejemplo de componente react de clase.

class ContadorDeTiempo extends React.Component {

constructor(props) {

// Las props son las propiedades dentro del componente

super(props);

// State guarda el estado actual del componente

this.state = { segundos: 0 };

}

aumentarSegundos() {

// El método `setSdtate` actualiza el estado del componente.

this.setState(state => ({

segundos: state.segundos + 1

}));

}

// El método `componentDidMount` es parte del ciclo de vida

// Y se ejecuta justo después de que se monte el componente.

componentDidMount() {

this.intervalo = setInterval(() => this.aumentarSegundos(), 1000);

}

// El método `componentDidMount` es parte del ciclo de vida

// Y se ejecuta justo después de que se monte el componente.

componentWillUnmount() {

clearInterval(this.intervalo);

}

// El método `render` regresa la definición de cómo se va a ver

// el componente en determinado estado del tiempo

render() {

// Regresa un `div` con el contador enlazado al estado

// Y con acceso a la propiedad que se le paso como atributo

return (

<div>

<h1>

{/\* Las props son inmutables \*/}

{this.props.titulo}

</h1>

<div>

{/\* El estado cambia \*/}

Segundos: {this.state.segundos}

</div>

</div>

);

}

}

// Renderiza el componente `ContadorDeTiempo`

// Y lo monta en el elemento HTML con id `ejemplo-componente`

ReactDOM.render(

<ContadorDeTiempo titulo="Ejemplo Contador" />,

document.getElementById('ejemplo-componente')

);

</script>

La arquitectura en la que está basada ReactJS otorga flexibilidad y comprensión natural del flujo de los datos entre cada componente, en el ejemplo se muestra la definición de un componente de clase usando métodos del ciclo de vida, el cual renderiza un div que actualiza los segundos que han pasado desde que se montó, por lo que cada 1000 milisegundos actualiza el estado. Así mismo, se puede observar que el método render() utiliza elementos HTML y los trata como si fueran Javascript, a esta forma de renderizar componentes se le denomina JSX syntax. Esta sintaxis no es obligatoria en React, pero hace la legibilidad del componente más entendible.

Estos atributos de React otorgan flexibilidad a los desarrolladores del mundo, razón por la cual es el frontend framework con mayor interés dentro de la comunidad durante varios años consecutivos. De acuerdo con el State of Javascript 2019 con un 89% de satisfacción en comparación con otros frameworks como VueJS, Svelte o Angular. Tomando en cuenta todas las características de React y siendo una de las librerías más usadas en el mundo, se puede afirmar que, al hacer uso de ésta, se provee de una flexibilidad extraordinaria.

**Props**

Abstracta y conceptualmente hablando, un componente de React es una función de Javascript, ya que pueden aceptar argumentos (que se llaman props) y devuelven representaciones de elementos que describen lo que aparece en pantalla.

Las props son los atributos que recibe un componente cuando se escribe por medio de JSX y a las cuales el componente tiene acceso para poder inicializarlo, se puede decir que las props son el estado inicial del componente, esto quiere decir que se puede pasar propiedades entre componentes dentro de componentes y hacia otros componentes, ya que la composición otorga esta flexibilidad. Sin embargo, al ser una representación inicial, estas propiedades no deben cambiar dentro del componente, ya que React refuerza el principio de inmutabilidad de la programación funcional y el uso de funciones puras para el renderizado de los componentes.

// Componente funcional, las props se reciben en el método.

function BienvenidoFuncional(props) {

return (<h1>Hola, {props.nombre}</h1>);

}

// Componente de clase, las props forman parte del objeto en sí.

class BienvenidoClase extends React.Component {

render() {

return <h1>Hola, {this.props.nombre}</h1>;

}

}

// Una aplicación de React usa un único componente App para renderizar

// toda la aplicación, desde donde se montan todos los demás componentes

function App() {

// Las props se inicializan por medio de los atributos del componente

// Los dos tipos de componentes son válidos y React los renderiza igual

return (

<div>

<BienvenidoFuncional nombre="Armando" />

<BienvenidoClase nombre="Celina" />

<BienvenidoFuncional nombre="Luis" />

<BienvenidoClase nombre="Sofia" />

</div>

);

}

// Se renderiza la aplicación sobre el elemento con `id="root"`

ReactDOM.render(

<App />,

document.getElementById('root')

);

*Ejemplo de componente funcional y de clase con props en React*

**Si deseas consultar y descargar una infografía que resume un poco lo visto hasta ahora de Fundamentos de REACT**[**presiona aquí**](https://github.com/U-Camp/BOOT-M2-SEM13/blob/main/infografias/M2_S13_fundamentosreact.pdf)

**Sigamos adelante con State.**

**State**

El estado de un componente es similar a las props, pero en lugar de que el estado sea transferido al componente, el estado es privado y no es inmutable, por lo que un componente puede cambiar su estado dependiendo de las interacciones y funcionalidades internas del componente, por lo que es el mismo componente el que se encarga de manejar su propio estado.

Al final, el estado es la representación de los datos de un componente en un tiempo determinado, estos datos son mutables y pueden cambiar durante el uso de la aplicación. El estado no es obligatorio, lo que da pie a que existan Stateful Components (componentes con estado) y Stateless Components (componentes sin estado).

Para poder asignar un estado a un componente, se asigna como cualquier otro atributo dentro de la clase, en el constructor usando un this.state, el cual va a guardar el estado del componente y se le puede asignar cualquier valor necesario para que el componente realice su funcionalidad. Al usar React un paradigma funcional para el manejo de los atributos de los componentes, no se puede modificar el estado directamente, la única vez que se asigna es en el constructor, pero cualquier otra modificación que se requiera hacer sobre el estado, se debe realizar usando el método especial setState, el cual forma parte de react y es encargado de realizar y manejar de manera adecuada las actualizaciones de estado al estar internamente ligado al algoritmo de reconciliación que realiza el renderizado del DOM.

// Creamos un componente funcional sin estado

function FechaEnFormato(props) {

return <h2>La fecha y hora actual es: {props.fecha.toUTCString()}.</h2>;

}

// Creamos un componente de clase con estado

class Reloj extends React.Component {

constructor(props) {

super(props);

// Asignamos el estado inicial del componente

this.state = {

fecha: new Date()

};

// Actualizamos el estado cada segundo

setInterval(

() => this.setState({ fecha: new Date() }),

1000

);

}

// Mostramos la fecha y la hora actuales

// lo cual está enlazado al estado del componente.

// El estado de un componente puede pasarse

// como prop a otro componente.

render() {

return (

<div>

<FechaEnFormato fecha={this.state.fecha} />

</div>

);

}

}

// Se renderiza el componente sobre el elemento con `id="root"`

ReactDOM.render(

<Reloj />,

document.getElementById('root')

);

**Ciclos de vida**

Cuando un componente es Stateful, éste tiene acceso a determinados métodos de su propio ciclo de vida:

* constructor(): No es considerado un método del ciclo de vida como tal, sino como un método inherente al objeto al que hace referencia el componente, no se usa más que para instanciar un estado inicial del componente o para enlazar métodos al contexto del objeto.
* render(): Tampoco es considerado un método del ciclo de vida como tal, pero todo componente (sea Stateful o Stateless) lo debe tener, éste método siempre regresa la definición de cómo se va a ver el componente en determinado estado del tiempo ya que éste método es inmutable y tiene que ser una función pura de Javascript.
* componentDidMount(): Este método se ejecuta si el componente se montó en el Virtual DOM correctamente, justo después de render(), normalmente sirve para causar side effects en la interfaz, los cuales no podemos realizar en render(), ya que éste método debe ser puro. Un side effect común es obtener datos de una API y actualizar el estado del componente con base en esos datos.
* componentWillUnmount(): Este método es ejecutado cuando el componente se desmonta del Virtual DOM, normalmente se usa para realizar una limpieza de todos aquellos recursos que los side effects hayan dejado abiertos (como algún socket abierto o algún stream que se haya quedado transmitiendo).

Existen otros métodos del ciclo de vida (como getDerivedStateFromProps(), shouldComponentUpdate() o getSnapshotBeforeUpdate()), pero no son ampliamente usados ya que se usan sólo en casos específicos donde la funcionalidad interna de ReactJS necesite ser extendida.

En últimas versiones de ReactJS, los métodos del ciclo de vida y del manejo del estado se han ido deprecando o en su defecto han sido reemplazados por los hooks, los cuales son métodos específicos de cada componente que intentan resolver problemas de lógica y de entendimiento humano / máquina que los componentes de clase generan, el hook que reemplaza los métodos del ciclo de vida es el useEffect(), el cual desencadena un side effect y dependiendo de un arreglo de dependencias que recibe como parámetros, éste hook se ejecuta en determinada parte del ciclo de vida.

Para el manejo de estado tenemos el useState(), el cual devuelve un arreglo con una variable de estado en la primera posición y un método para actualizar esa variable en la segunda posición. Existen otros hooks que son menos usados como el useContext(), useMemo() o useCallback(), se implementan únicamente si la funcionalidad los necesita. Todos estos hooks así como su definición e implementación se revisarán a profundidad posteriormente.

En el ejemplo del componente Reloj de la clase anterior, se inicializaba el método que actualiza el estado directamente en el constructor, y aunque esto no es una mala práctica como tal, ocasiona un side effect el cual es mejor controlar dentro del método del ciclo de vida componentDidMount ya que esos métodos fueron hechos para tal fin, así como también hay que liberar la memoria y limpiar el intervalo de actualización una vez se haya desmontado el componente con el método componentWillUnmount, por lo que el ejemplo del reloj refactorizado para usar métodos del ciclo de vida quedaría de la siguiente manera:

// Creamos un componente funcional sin estado

function FechaEnFormato(props) {

return <h2>La fecha y hora actual es: {props.fecha.toUTCString()}.</h2>;

}

// Creamos un componente de clase con estado

class Reloj extends React.Component {

constructor(props) {

super(props);

// Asignamos el estado inicial del componente

this.state = {

fecha: new Date()

};

}

// Manejamos el side effect

componentDidMount() {

// Guardamos en la clase el identificador del intervalo

// esto sirve para poder limpiarlo antes de desmontar el componente

this.idIntervalo = setInterval(

() => this.actualizarFecha(),

1000

);

}

// Antes de que el componente se desmonte, limpiamos el intervalo

componentWillUnmount() {

clearInterval(this.idIntervalo);

}

// Método custom para asignar el estado en cada ciclo del intervalo

actualizarFecha() {

this.setState({ fecha: new Date() });

}

// Mostramos la fecha y la hora actuales

// lo cual está enlazado al estado del componente.

// El estado de un componente puede pasarse

// como prop a otro componente.

render() {

return (

<div>

<FechaEnFormato fecha={this.state.fecha} />

</div>

);

}

}

// Se renderiza el componente sobre el elemento con `id="root"`

ReactDOM.render(

<Reloj />,

document.getElementById('root')

**Eventos**

Manejar eventos en React es prácticamente lo mismo que manejar eventos en el DOM, sólo que el atributo para asignar el manejador del evento se define en camelCase y el cual recibe la referencia directa del método que manejará el evento en sí, básicamente este método es una función en Javascript que realiza determinadas acciones o side effects por lo que es igual que cualquier otro manejador.

// Componente de ejemplo de la clase anterior con eventos

function Contador(props) {

// Arreglo con variable de estado - método de actualización

const [contador, setContador] = useState(0);

// Método que manejará el evento de incremento del contador

const manejarIncremento = () => {

// Calculamos el nuevo valor

const nuevoValor = contador + 1;

// Lo asignamos al estado

setContador(nuevoValor);

};

// Método que manejará el evento de decremento del contador

const manejarDecremento = () => {

// Calculamos el nuevo valor

const nuevoValor = contador - 1;

// Lo asignamos al estado

setContador(nuevoValor);

};

// Los botones usan el atributo `onClick` para manejar el evento

return (

<div>

<h1>Hola, {props.nombre}</h1>

<p>Valor del contador: {contador}</p>

<button onClick={manejarIncremento}>

Sumar 1

</button>

<button onClick={manejarDecremento}>

Restar 1

</button>

</div>

);

}

*Ejemplo de manejo de eventos en react*

**Error boundaries**

Los Error Boundaries son componentes de React que capturan errores de Javascript en cualquier parte de su árbol de componentes hijo, registran esos errores, y muestran una interfaz de repuesto en lugar del árbol de componentes que ha fallado. Los error boundaries capturan errores durante el renderizado, en métodos del ciclo de vida, y en constructores de todo el árbol bajo ellos, sin embargo, cabe aclarar que los error boundaries no capturan errores de algún manejador de eventos, de código asíncrono o errores lanzados en el propio error boundary, todos esos tipos de errores se deben manejar con un bloque try … catch de preferencia, ya que son errores en tiempo de ejecución, no en tiempo de renderizado que son los errores que maneja el error boundary.

Para implementar un error boundary basta con crear un componente de clase que implemente los métodos del ciclo de vida static getDerivedStateFromError y componentDidCatch, los cuales se usan para renderizar la interfaz de repuesto en caso de error y para registrar la información de los errores lanzados respectivamente, una vez que se tenga este componente, se puede usar como High Order Component de tal manera que envuelva a todos los componentes que podrían causar un error en tiempo de renderizado. Si no se usan error boundaries y por alguna causa fortuita la interfaz se rompe y causa un error en tiempo de renderizado, toda la aplicación de React se desmonta y se rompe completamente, por lo que es recomendable siempre usar al menos un error boundary que envuelva todo el árbol de componentes de la aplicación y que permita recuperarse de algún posible error que pudiera ocurrir.

// Definimos el componente que servirá como error boundary

// Forzosamente tiene que ser un componente de clase

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

// Puede tener su propio estado para indicar si hubo algún error.

this.state = { error: null, errorInfo: null };

}

// Registra y tiene acceso al stack trace de los errores

componentDidCatch(error, errorInfo) {

// Captura algún error y setea el estado del árbol de componentes a erróneo

this.setState({

error: error,

errorInfo: errorInfo

});

}

render() {

// Si hubo algún error, renderiza una interfaz mostrando el error.

if (this.state.errorInfo) {

return (

<div>

<h2>Hubo un error :(.</h2>

<details style={{ whiteSpace: 'pre-wrap' }}>

{this.state.error && this.state.error.toString()}

<br />

{this.state.errorInfo.componentStack}

</details>

</div>

);

}

// Si no ha habido ningún error, no hay problema y renderizamos los componentes hijos

return this.props.children;

}

}

// Definimos un componente funcional con bugs

const ContadorConBugs = (props) => {

const [contador, setContador] = useState(0);

const manejarClick = () => setContador(contador + 1);

if (contador === 5) {

// Simulamos un error

throw new Error('Valió!');

}

// Si aún no hay error, renderizamos el contador

return <h1 onClick={manejarClick}>{contador}</h1>;

};

// Simulamos un árbol de componentes en una app

function App() {

return (

<div>

<p>

<b>

Ejemplo de error boundaries

<br /><br />

Haz click en los números para incrementar el contador

<br />

Si algún contador alcanza el número 5 se lanza un error simulado.

</b>

</p>

<hr />

<ErrorBoundary>

<p>

Estos dos contadores están envueltos en el mismo error boundary.

Si uno de los dos lanza un error, el error boundary reemplaza a ambos componentes.

</p>

<ContadorConBugs />

<ContadorConBugs />

</ErrorBoundary>

<hr />

<p>

Estos dos contadores están cada uno envueltos en su propio error boundary.

Por lo que si uno lanza un error, el otro no se ve afectado.

</p>

<ErrorBoundary><ContadorConBugs /></ErrorBoundary>

<ErrorBoundary><ContadorConBugs /></ErrorBoundary>

</div>

);

}

// Renderizamos la app

ReactDOM.render(

<App />,

document.getElementById('root')

);

*Ejemplo de error boundaries en React*

**Muy bien, has finalizado el contenido de esta semana, nos vemos en la clase del sábado.**

**Si presentas alguna duda recuerda que tienes a tus coaches a quiénes les puedes consultar.**

**Nos vemos pronto.**