CDN para produccion

<script crossorigin src="https://unpkg.com/react@17/umd/react.production.min.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.production.min.js"></script>

CDN para desarrollo

<script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>

La principal diferencia entre estos dos, es que puede que el cdn producción de error al manejar ciertos valores. Se recomienda usar el de desarrollo

Si se realiza un **console.log()** de **React** y **ReactDOM** se puede visualizar que ambos son objetos y cuentan con sus métodos y propiedades.

Estos dos CDN corresponden a React y ReactDOM.   
React trabaja con un “Virtual DOM” que hace que la generación y carga de elementos, sea más liviana ya que genera los elementos y no un código fuente de los mismos.

React trabaja con renderizados.   
Puede comprobarse que al crear un elemento desde react, este mismo no se visualiza en el código fuente de la página.

Un ejemplo de insertar un elemento con react es el siguiente:

        ReactDOM.render(

            React.createElement("h1", null, "Hi World"),

            root);

se invoca el objeto ReactDOM y se utiliza su método **.render**. Este método recibe 2 parámetros, el primero el elemento y el segundo la ubicación en la cual se renderizará.

Para el primero, se invoca el objeto React con su método createElement y se crea un h1 con texto “Hi World”. Para el segundo se selecciona root, que es un div#root.

El segundo parámetro corresponde a las propiedades, aunque puede ir en **null.**

El tercer parámetro corresponde a los “children”, en este caso, al texto del elemento.

**PROBLEMAS SIN JSX**

Todas las creaciones de un elemento que se puedan hacer de esta forma, son con React puro, sin usar JSX. El problema es que cuando se necesiten crear, por ejemplo, <ul> con varios <li> dentro y a su vez otros <ul> con <li>, se vuelve difícil de manejar de esta forma.

**BABEL Y JSX**

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

Una opción de compilador de texto JS es BABEL. Este nos permite crear elementos React desde HTML con mas facilidad, utilizando código XML.

Para poder utilizar este, es necesario que el atributo type= de la etiqueta <script> tenga como valor “text/label”, además de haber incorporado el cdn mediante la etiqueta indicada anteriormente.

Un ejemplo de comparación de código entre React y Babel es el siguiente:

Con react:   
        ReactDOM.render(

            React.createElement("ul", null, React.createElement('li', null, "Anakin"),

                React.createElement('li', null, "Obi Wan"),

                React.createElement('li', null, "Padme"),

                React.createElement('li', null, "Mace Windu"),

                React.createElement('ol', null, React.createElement('li', null, 'ironMan'),

                    React.createElement('li', null, 'Cap'),

                    React.createElement('li', null, 'thor'))),

            root);

se hace poco sostenible, además de que algunos resultados se vuelven muy complejos.

Con babel:

    <script type="text/babel">

        ReactDOM.render(

        <div>

            <h2> creando la lista con babel</h2>

            <ul>

                <li> Anakin </li>

                <li> Obi-wan </li>

                <li> Padme </li>

                <li> Mace Windu </li>

                <ol>

                    <li>Iron Man</li>

                    <li>Cap</li>

                    <li>Thor</li>

                </ol>

            </ul>

        </div>

        , document.querySelector('#root'))

    </script>

Practicamente es escribir HTML.

**INTRODUCCION A LOS COMPONENTES**

Los componentes son como “partes” de la aplicación. Lo que se busca con estos, es modular la aplicación, en todos los sentidos. Tanto como para la integración de nuevos componentes, como para su lógica y funcionamiento.   
Generalmente, cuando los componentes son muy complejos, se crea un archivo dedicado a cada uno.  
Esto, hace que se puedan crear sitios grandes o aplicaciones grande, integrando varios componentes pequeños.

**FORMA ANTIGUA DE DECLARAR COMPONENTES**

Anteriormente se declaraban los componentes como clases, como la POO

        class Saludo extends React.Component{

            render(){

                return (

                <h2>asd</h2>

                );

            }

        }

En este caso se crea una clase Saludo, que extiende la clase principal del objeto React, que corresponde a los componentes y en su método **render()** retorna un elemento <h2>

Para “imprimir” este era necesario

ReactDOM.render(<Saludo></Saludo>, document.querySelector('#paraSaludos'))

Se crea una “Nueva etiqueta” con el nombre de la clase y se inserta en el div#parasaludos

**DECLARANDO COMPONENTES DE LA MANERA MODERNA**

Para implementar este nuevo modo, se procede de forma muy similar a como se trabajo en la nivelación de JS, con const.

        const Saludito = () => { return (

        <h2> Creando elemento vanguardista</h2>); } ; ReactDOM.render(

        <Saludito></Saludito>, document.querySelector('#vanguar'));

Se crea una constante Saludito que es una arrow function, está retorna un <h2>  
luego mediante ReactDOM.render() y con la “nueva etiqueta” se introduce el elemento en un div#vanguar

Nota: cuando se aplica la etiqueta “nueva” como primer parámetro de ReactDOM.render() puede establecerse como etiqueta **self-closing**

**INTERPOLANDO VARIABLES**

Para insertar contenido dinámico en un componente, se cumple el mismo principio de los templates strings, se busca un grupo de caracteres en los cuales se ejecute código JS.

        const name = prompt('introduce tu nombre');

        const Interpolar = () => {

        return(

            <div>

                <h3>Interpolando variables like a pro</h3>

                <h2>Welcome {name}</h2>

                <hr/>

            </div>

        ); };

        ReactDOM.render(<Interpolar/>, document.querySelector('#inter'));

En este caso, se utiliza como template {}. Todo lo que esté dentro de {} se ejecuta como código de JS. Para el ejemplo, se guarda en una const el valor recibido por prompt() y se imprime junto con el elemento <Interpolar> </Interpolar>

**INTRODUCCION A LOS ESTADOS**

Los estados de los componentes o aplicaciones son los valores que contienen estos en determinado momento. No deberían ser manipulados manualmente por los programadores ya que es un “tema delicado”.

Se “actualizan” cada vez

Los estados se manejan con el objeto **React.useState**. Este es un ejemplo completo:

      const Fecha = () => {

        const [numero, setNumero] = React.useState(0);

        setTimeout(() => {

          setNumero(numero + 15);

        }, 3000);

        return;

        <h3>El numero es: {numero}</h3>;

      };

      ReactDOM.render(<Fecha />, document.querySelector("#paraStates"));

En la constante fecha se guarda un arrow function, cuyo retorno es un h3 que contiene un “template”

Dentro de la funcion, se declara una constante que es un array que contiene **numero** y la funcion **setNumero**. En React.useState se pasa por parámetro el “estado inicial”.

Luego en setTimeout se modifica este estado, para luego retornar su valor en el **return**  de la funcion.

Por ultimo se renderiza la etiqueta en un div#paraStates.

Es importante recordar que por alguna extraña razón, las etiquetas comienzan sus nombres con Mayusculas.

Además, a pesar de que esto pareza ejecutarse solo una vez, el estado se actualiza constantemente.  
Se podría decir que cuando se invoca el render, se tiende a actualizar cada cierto tiempo, y al insertar el timeout o Interval, se fuerza a que se ejecute a ese tiempo.