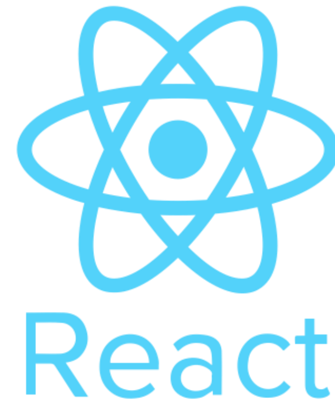
React JS



Web: https://reactjs.org/

React es una librería Front-End desarrollada por Facebook que nace por un problema de rendimiento que sufrió la red social Facebook y que produjo la creación de esta librería que venía a mejorar el renderizado gracias a su forma de trabajar mediante el Virtual DOM.

Características:

Promueve la construcción de interfaces modulares.

Favorece la reutilización de componentes.

Se pueden usar las características modernas de JavaScript.

Velocidad de renderizado.

Flujo de datos es unidireccional (Componentes padres e hijos).

Tiene una gran comunidad de desarrolladores.

Arquitectura

React hace uso del paradigma **"**Programación orientada a componentes", en otras palabras, las interfaces que se crean con React están basadas en diferente partes llamadas componentes los cuales son dinámicos y reutilizables, donde los mismos encapsulan su propia lógica y presentación mediante los ficheros JSX.

Virtual DOM

Es una representación del DOM de la aplicación en memoria que detecta los cambios de estado de un componente que forma el DOM y por lo que renderiza solo esa parte del DOM con lo que el proceso de renderizado de la aplicación es mucho más eficiente, ya que no tiene que renderizar todo el DOM

En definitiva, React realiza una copia de tu aplicación web y la guarda en memoria, cuando se realiza una acción sobre un componente que cambia el estado, React escucha el cambio y lo compara con la copia que tiene en memoria, entonces actualiza el componente sin necesidad de refrescar el navegador.

La principal característica de este "Virtual DOM", a diferencia del DOM común, es que es muy ligero.

Componente

Elemento del árbol de la aplicación, reutilizable, que cumple un objetivo y una función independiente.

Los componentes pueden ser:

Funcionales: Definidos a través de function, retornan un elemento, pueden tener propiedades como argumentos, pero no poseen estado ni ciclo de vida.

```
const NombreComponente = () => (
                               <div>
                                 <h1>Hola soy un componente function</h1>
                               </div>
                               );
```

Basados en clases ES6: Definidos a través de class, heredan de React.Component, se puede declarar con constructor(), es necesario definir el método render(), se puede controlar el ciclo de vida y declarar lógica para eventos.

```
class NombreComponente extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return (
            <div>
                <h1>Hola soy un componente de clase</h1>
            </div>
        )
    }
}
```

Puros: También se definen a través de class, pero heredan de React.PureComponent, su diferencia reside en que no implementan el método shouldComponentUpdate() por lo que no podemos saber si el componente se tiene o no que renderizar de nuevo, estos componentes tienen un mayor rendimiento, pero son recomendados con componentes simples que devuelvan la misma salida para los mismos estados y propiedades.

```
class NombreComponente extends React.PureComponent {
    const styles = { border: '1px solid blue', color: 'white' }

    render() {
        return (
            <p {...this.props} style={styles}>Párrafo</p>
        );
    }
}
```

JSX (JavaScript XLM)

Es la sintaxis que usa React para construir los componentes en el virtual DOM, a través de este código JSX que se transcompila a JavaScript mapeando los elementos del DOM para crear una representación en memoria llamada "Virtual DOM", a partir de la cual luego crea el DOM definitivo en el navegador.

Algunas características de esta sintaxis son:

El atributo class de HTML es una palabra reservada de JavaScript, por lo que en JSX se utiliza className

Las variables en JSX se ponen mediante llaves {miVariable}

El método render() debe devolver un único elemento HTML

```
class NombreComponente extends React.Component {
    render() {

        const miVariable = "Hola soy un componente"

        return (
            <div>
                <p className="nombreClase" >{miVariable}</p>
            </div>
        );
    }
}
```

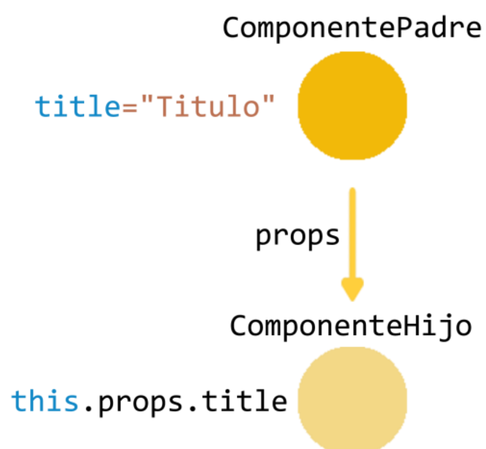
Props

Las props son la manera en que un componente padre manda información a componentes hijo, solo el padre las puede configurar (al instanciar el componente) y modificar.

Las propiedades son inmutables se deben de utilizar en modo lectura.

```
class ComponentePadre extends React.Component {

    constructor(props) {
        super(props);
    }

    render() {
        return (
            <div>
                <ComponenteHijo title="Titulo" />
            </div>
        );
    }

}
```

```
class ComponenteHijo extends React.Component {

    render() {
        return (
            <div>
                <h1>{this.props.title}</h1>
            </div>
        );
    }

}
```



State

El state es una representación del componente en un punto en el tiempo, se inicia con unas características propias y puede mutar durante el ciclo de vida del componente, cambiando así, su estado y renderizando nuevamente el componente por completo

state se inicializa y configura en el constructor() del componente

La definición de estados en un componente es opcional.

Para modificar el estado de un componente debemos de usar .setState()

```
class ComponenteParrafo extends React.Component {

    constructor() {
        super();
        this.state = {
            parrafo: "Texto Inicial"
        };
        this.updateParrafo = this.updateParrafo.bind(this);
    }

    updateMessage() {
        this.setState({
            parrafo: "Texto Modificado"
        });
    }

    render() {

        return (
            <div>
                <p>{this.state.parrafo}</p>
                <button onClick={this.updateParrafo}>
                    Click para actualizar el texto
                </button>
            </div>
        )
    }
}
```

Ciclos de vida de los componentes

Los componentes basados en clases, pasan por varias etapas llamadas ciclos de vida, que son funciones especiales que nos permiten ejecutar lógica de manera más controlada.

Los ciclos de vida de los componentes son: montado, actualización y desmontado.

1.Montado

Primera fase de vida del componente, el componente se crea y ocurre una única vez.

1.1 constructor(props) {}

Define la configuración inicial del componente

Se ejecuta al instanciar la clase

Inicializa las variables que usa en la clase

```
class MiComponente extends React.Component {

    constructor(props) {

        super(props);

        this.state = {
            texto: "Texto",
        };

        this.miVariable = true;
    }
}
```

1.2 componentWillMount() "deprecated"

Se llama antes de que el componente sea montado

Se pueden cambiar estados

Las modificaciones en este ciclo de vida no causan actualizaciones en el componente

No se deben realizar suscripciones a eventos (No existe ni window ni document) ni llamadas a APIs

```
class MiComponente extends React.Component {

    constructor(props) { }

    componentWillMount() {
        this.setState({ texto: "Texto a mostrar" });
    }

}
```

1.3 render () {}

Genera la interfaz gráfica inicial

Contiene la estructura base a renderizar y se muestra en el Front-End

No se debe realizar suscripciones a eventos ni modificar el estado, puede causar un ciclo infinito

```
class MiComponente extends React.Component {

    constructor(props) { }
    componentWillMount() { }

    render() {

        return (
            <div>
                <p>{this.state.texto}</p>
            </div>
        );

    }

}
```

1.4 componentDidMount() {}

Se ejecuta cuando el componente se encuentra montado (solo se llama una vez)

Se realizan las suscripciones a eventos, llamadas a APIs y setTimeout, ya que existe window y document

```
class MiComponente extends React.Component {
    constructor(props) { }
    componentWillMount() { }
    render() { }

    componentDidMount() {
        fetch('/textos/getTextos')
            .then(response => response.json())
            .then(textos => {
                this.setState({ texto: textos[0] });
            });
    }

}
```

2.Actualizacion

Los cambios de estado y propiedades, son los causantes de estas actualizaciones en los componentes, estos pueden o no actualizarse y hacerlo más de una vez.

2.1 componentWillReceiveProps(nextProps) "deprecated"

Se ejecuta al recibir un cambio en las propiedades del componente, realiza cambios en los estados basados en las nuevas propiedades.

```
class MiComponente extends React.Component {

    constructor(props) { }
    componentWillMount() { }
    render() { }
    componentDidMount() { }

    componentWillReceiveProps(nextProps) {
        this.state({
            texto: nextProps.texto
        });
    }

}
```

2.2 shouldComponentUpdate(nextProps, nextState) {}

Permite indicar si nuestro componente tiene que renderizar de nuevo, devolvemos true o false.

Es usado para optimizar el rendimiento.

```
class MiComponente extends React.Component {

    constructor(props) { }
    componentWillMount() { }
    render() { }
    componentDidMount() { }
    componentWillReceiveProps(nextProps) { }

    shouldComponentUpdate(nextProps, nextState) {
        return (this.props.texto !== nextProps.texto);
    }

}
```

2.3 componetWillUpdate(nextProps, nextState) "deprecated"

Se invoca antes de realizar un nuevo renderizado del componente, pero solo se ejecuta cuando shouldComponentUpdate(nextProps, nextState) nos devuelva true y se realizan los últimos cambios antes de renderizar de nuevo.

Es usado para optimizar el rendimiento.

```
class MiComponente extends React.Component {

    constructor(props) { }
    componentWillMount() { }
    render() { }
    componentDidMount() { }
    componentWillReceiveProps(nextProps) { }
    shouldComponentUpdate(nextProps, nextState) { }

    componentWillUpdate(nextProps, nextState) {
        if (this.state.texto !== nextStage.texto) {
            this.cambiarColor();
        }
    }

}
```

2.4 componentDidUpdate(prevProps, prevState, snapshot) {}

Se ejecuta después de ser de nuevo renderizado el componente

```
class MiComponente extends React.Component {

    constructor(props) { }
    componentWillMount() { }
    render() { }
    componentDidMount() { }
    componentWillReceiveProps(nextProps) { }
    shouldComponentUpdate(nextProps, nextState) { }
    componentWillUpdate(nextProps, nextState) { }

    componentDidUpdate(prevProps, prevState, snapshot) {
        this.adjuntarEnlace();
    }

}
```

3.Desmontado

Última fase de vida de los componentes

3.1 componentWillUnmount() {}

Su función es limpiar el componente, este se ejecuta antes de que el componente haya sido desmontado y su misión es dejar de escuchar eventos y cancelar peticiones http pendientes.

```
class MiComponente extends React.Component {

    constructor(props) { }
    componentWillMount() { }
    render() { }
    componentDidMount() { }
    componentWillReceiveProps(nextProps) { }
    shouldComponentUpdate(nextProps, nextState) { }
    componetWillUpdate(nextProps, nextState) { }
    componentDidUpdate(prevProps, prevState, snapshot) { }

    componentWillUnmount() {
        window.removeEventListener("resize", this.getDimensiones());
    }

}
```

Nota: Los métodos marcados como deprecated se renombrarán y se introducirán dos nuevos métodos para la versión 17

UNSAFE_componentWillMount

UNSAFE_componentWillRecieveProps

UNSAFE_componentWillUpdate

getDerivedStateFromProps

getSnapshotBeforeUpdate

Exportar e importar componentes

Creamos el componente y añadimos un export al final.

```
class MiComponente extends React.Component {}

export default MiComponente;
```

Después realizamos un import al inicio de la nueva clase donde queremos usarlo.

```
import MiComponente from './MiComponente';
```