	UNIVERSIDAD DON BOSCO **FACULTAD DE INGENIERÍA** **ESCUELA DE COMPUTACIÓN**	
CICLO: 02	**GUIA DE LABORATORIO #02**	
	Nombre de la Práctica:	**React , parte I**
	MATERIA:	**Diseño y Programación de Software Multiplataforma**

I. OBJETIVOS

Que el estudiante:

- Diseñe aplicaciones web utilizando funciones de React.
- Diseñe aplicaciones con navegación.
- Aprender hacer interfaz para que el usuario pueda interactuar con la aplicación.
- Aprender a crear componente personalizado.
- Aprender hacer interfaz para que el usuario pueda interactuar con la aplicación.
- Aprender a utilizar pipes dentro de la interfaz de usuario.
- Hacer uso de los componentes en el diseño de interfaz React.

Contenido

II. INTRODUCCION TEORICA

¿Qué es React JS?

Se trata de una librería de Javascript creada por Facebook que nos ayuda a la hora de desarrollar aplicaciones en una sola página. Es una biblioteca de código abierto con una comunidad muy activa que comparte componentes que todo desarrollador podrá usar en sus proyectos.

Nota: Si quieres desarrollar webs apps y quieres aprender bien desde cero, te recomendamos el curso de desarrollo de aplicaciones web con ReactJS.

¿Qué hace diferente a React JS de otras librerias?

Utiliza lenguaje JSX para construir la interfaz de usuario, por lo que si dominas HTML podrás con gran facilidad crear tus propios componentes y su comportamiento. JSX crea plantillas donde se pueden anidar elementos, por lo que podrás crear componentes tan pequeños o grandes como creas conveniente.

React JS usa un DOM Virtual que recarga individualmente cada componente cuando haga falta, por lo que ofrece una gran velocidad a la hora de renderizar vistas.

Los componentes son totalmente independientes y contienen su propio comportamiento, un estado y el contenido a renderizar. Éstos, contienen elementos que se pueden anidar, igual que podemos anidar componentes y que se comuniquen entre ellos.

Las aplicaciones en React JS tienen un estado global que repercute en el estado individual de cada componente.

El isomorfismo (renderización HTML en servidor como en cliente) de React JS ofrece la posibilidad de entregar el HTML ya renderizado a los buscadores web, como Google, y así mejorar el posicionamiento. De esta manera se evita el problema de que el buscador se encuentre el cuerpo de la página vacío).

Creación de un Componente y Hola Mundo

Como hemos comentado más arriba, la base de React JS son los **componentes**. Es por ello que es imprescindible comprender como crear uno y como renderizarlo en nuestro proyecto.

Cuando desarrollamos componentes de React JS, debemos tratar de individualizarlos lo máximo posible para que podamos usarlos en otros proyectos, ya sean existentes o nuevos.

En esta ocasión vamos a crear un componente llamado (Saludo.js) - en React, los componentes deben comenzar con mayúscula - que renderizará el famoso mensaje de Hola Mundo.

Crearemos Saludo.js a la misma altura de carpetas que App.js para simplificar, dejaremos el header con el logo de React y únicamente cambiaremos el cuerpo de App.js añadiendo una **etiqueta <Saludo />** que será nuestro nuevo componente.

Saludo.js
Haciendo uso de **ES6**, definiremos Saludo.js como una **clase** que va a heredar todo aquello que tiene un componente React (**Component**).

El método render de la clase Saludo, básicamente devuelve un **JSX** con un título cuyo contenido es **Hola Mundo**. También tenemos definido un constructor donde se define el estado del componente y donde podremos acceder y definir las propiedades y comportamiento del componente.

Para poder hacer uso de este componente en App.js y renderizarlo, debemos exportarlo (export default) con un nombre. En este caso Saludo.

```
import React, { Component } from 'react';

class Saludo extends Component {
    render() {
        return (<h1> Hola Mundo! </h1>);
    }
} export default Saludo;
```

App.js
El **componente principal de todo proyecto** que use React JS es **App.js** y es el encargado de contener todos los componentes necesarios para la aplicación, ya sean propios o externos. En nuestro caso, App.js sera el encargado de renderizar un componente Saludo para que así podamos ver por pantalla "Hola Mundo".

Para poder hacer uso de la etiqueta <Saludo /> debemos importarlo al comienzo del fichero. Debes editar el archivo para que quede de la siguiente manera:

```
import React,
    {
    Component
}

from 'react';
import Saludo from './Saludo'import logo from './logo.svg';
import './App.css';
```

```
class App extends Component {
    render() {
        return (<div className="App"> <header className="App-header"> <img src= {
                logo
            }

            className="App-logo"alt="logo"/> <h1 className="App-title">Welcome to Re
act</h1> </header> <Saludo /> </div>);
    }
}

export default App;
```

index.js

En este archivo se define cómo incrustar nuestro componente principal (App), que contendrá en su interior el componente Saludo que hemos creado un poco mas arriba. Haciendo uso de la clase **ReactDOM**, podremos renderizar App en nuestro unico fichero html: **index.html**.

Concretamente renderizará App en el elemento HTML que contenga el id "root".

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(
    <App />,
    document.getElementById('root'));
registerServiceWorker();
```

index.html

Este es el HTML de nuestro proyecto que contendrá el componente App y éste a su vez, contendrá el nuevo componente Saludo. Puedes apreciar el elemento donde se va a introducir, ya que dispone del id root.

Resultado

Los cambios que vayas haciendo en cualquiera de los archivos o carpetas, se ven rápidamente reflejados en el navegador sin la necesidad de re-desplegar manualmente. Una vez hemos realizado nuestro componente Saludo y lo hemos introducido en el componente App, vamos a ver cómo queda:

Hola Mundo!

JSX

Como ya hemos visto, todos los componentes de React tienen una función de render. La función de render especifica la salida HTML de un componente React.

JSX (Extensión JavaScript), es una extensión de React que permite escribir código JavaScript que se parece a HTML.

En otras palabras, JSX es una sintaxis similar a HTML utilizada por React que extiende ECMAScript para que la sintaxis similar a HTML pueda coexistir con el código JavaScript / React.

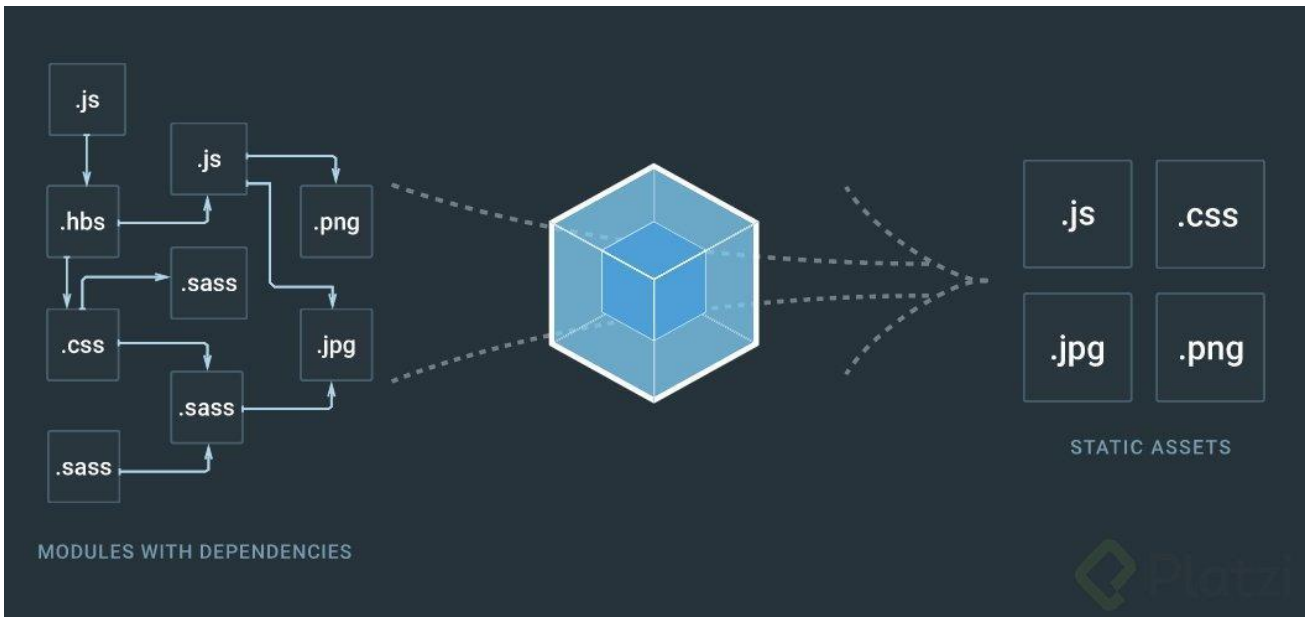
La sintaxis es utilizada por preprocesadores (es decir, transpiladores como babel) para transformar la sintaxis similar a HTML en objetos JavaScript estándar que analizará un motor JavaScript.

JSX le permite escribir estructuras similares a HTML / XML (por ejemplo, estructuras de árbol similares a DOM) en el mismo archivo donde escribe el código JavaScript, luego el preprocesador transformará estas expresiones en código JavaScript real. Al igual que XML / HTML, las etiquetas JSX tienen un nombre de etiqueta, atributos e hijos.

Webpack

Es un Module Bundler for modern JavaScript applications, es decir, un empaquetador de módulos para aplicaciones modernas hechas en JavaScript.

Una aplicación web lleva múltiples tipos de assets como imágenes, fuentes personalizadas, JSON, CSS, JavaScript, etc. y manejar esto se vuelve un dolor de cabeza a medida que nuestra aplicación tiene éxito y va creciendo. Todo esto lo resuelve Webpack y es por eso que será nuestro empaquetador de módulos (o Module Mundler) ya que para él, todos estos archivos serán tratados de esta manera:



¿Por qué es importante aprender Webpack?

Tener una aplicación organizada va a ayudarte a ti y a tu equipo, a llevar un correcto manejo de versiones de tu código. También necesitarás organizar carpetas llenas de archivos en toda clase de extensiones y separar las que se usan en entornos de desarrollo, como las que se usarán en producción. Esto lo hace Webpack de manera fácil, solo debes decirle dónde está el archivo fuente y a dónde quieres enviar el o los archivos resultantes.

Developer experience

Hoy en día es muy difícil pensar en iniciar una aplicación sin primero plantearnos un Stack de tecnología para crearlo. Empiezan a sonar cosas como Angular, React, Vue para manejar nuestros componentes y PostCSS, Sass, Stylus o Less para manejar nuestros estilos.

Todo esto lleva un mundo detrás de código que el navegador no entiende de buenas a primeras porque seguro escribirás código en Typescript, templates en jsx o simplemente usar todo los nuevos features de **EcmaScript2017**. Con Webpack no tendrás que sacrificar tu comodidad a la hora de escribir código.

React Components

Los componentes son bits de código independientes y reutilizables. Tienen el mismo propósito que las funciones de JavaScript, pero funcionan de forma aislada y devuelven HTML a través de una función de representación.

Los componentes vienen en dos tipos, componentes de clase y componentes de función, en este tutorial nos concentraremos en los componentes de clase.

Crear un componente de clase

Al crear un componente de React, el nombre del componente debe comenzar con una letra mayúscula.

El componente tiene que incluir la extends React.Componentdeclaración, esta declaración crea una herencia a React.Component y le da a su componente acceso a las funciones de React.Component.

El componente también requiere un método render(), este método devuelve HTML.

Crea un componente de clase llamado Car

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

Ahora la aplicación React tiene un componente llamado Car, que devuelve un <h2>elemento.

Para utilizar este componente en su aplicación, utilice una sintaxis similar a la de HTML normal: <Car />

Muestre el Carcomponente en el elemento "raíz":

```
ReactDOM.render(<Car />, document.getElementById('root'));
```

Crear un componente de función

Este es el mismo ejemplo que el anterior, pero creado utilizando un componente de función en su lugar.

Un componente de función también devuelve HTML y se comporta de la misma manera que un componente de clase, pero los componentes de clase tienen algunas adiciones

Cree un componente de función llamado Car

```
function Car() {
  return <h2>Hi, I am also a Car!</h2>;
```

```
}
```

Una vez más, su aplicación React tiene un componente Car.

Consulte el componente Car como HTML normal (excepto en React, los componentes deben comenzar con una letra mayúscula):

Muestre el Carcomponente en el elemento "raíz":

```
ReactDOM.render(<Car />, document.getElementById('root'));
```

Constructor de componentes

Si hay una constructor() función en su componente, esta función se llamará cuando el componente se inicie.

La función constructora es donde inicia las propiedades del componente. En React, las propiedades del componente deben mantenerse en un objeto llamado state.

La función constructora también es donde honra la herencia del componente principal al incluir la super() declaración, que ejecuta la función constructora del componente principal, y su componente tiene acceso a todas las funciones del componente principal (React.Component).

Cree una función de constructor en el componente Coche y agregue una propiedad de color:

```
class Car extends React.Component {
 constructor() {
  super();
  this.state = {color: "red"};
 }
 render() {
  return <h2>I am a Car!</h2>;
 }
}
```

Utilice la propiedad de color en la función render ():

```
class Car extends React.Component {
 constructor() {
```

```
  super();
  this.state = {color: "red"};
 }
 render() {
  return <h2>I am a {this.state.color} Car!</h2>;
 }
}
```

Accesorios
Otra forma de manejar las propiedades de los componentes es usando **props**. Los apoyos son como argumentos de función y los envía al componente como atributos.

propsen

Use un atributo para pasar un color al componente Car y utilícelo en la función render ():

```
class Car extends React.Component {
 render() {
  return <h2>I am a {this.props.color} Car!</h2>;
 }
}
ReactDOM.render(<Car color="red"/>, document.getElementById('root'));
```

Componentes en componentes

Podemos referirnos a componentes dentro de otros componentes:

Utilice el componente Coche dentro del componente Garaje:

```
class Car extends React.Component {
 render() {
  return <h2>I am a Car!</h2>;
 }
}
class Garage extends React.Component {
 render() {
```

```
  return (
    <div>
    <h1>Who lives in my Garage?</h1>
    <Car />
    </div>
  );
 }
}
ReactDOM.render(<Garage />, document.getElementById('root'));
```

Componentes en archivos

React se trata de reutilizar código, y puede ser inteligente insertar algunos de sus componentes en archivos separados. Para hacer eso, cree un nuevo archivo con una .js extensión de archivo y coloque el código dentro de él:

Tenga en cuenta que el archivo debe comenzar importando React (como antes) y debe terminar con la declaración export default Car;.

Este es el nuevo archivo, lo llamamos "App.js":

```
import React from 'react';
import ReactDOM from 'react-dom';

class Car extends React.Component {
 render() {
  return <h2>Hi, I am a Car!</h2>;
 }
}
export default Car;
```

Para poder utilizar el componente Coche, debe importar el archivo en su aplicación.

Ahora importamos el archivo "App.js" en la aplicación y podemos usar el componente Car como si se hubiera creado aquí.

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
import Car from './App.js';

ReactDOM.render(<Car />, document.getElementById('root'));
```

IV. PROCEDIMIENTO

Para la primera aplicación en React, se necesitaran realizar algunas instalaciónes, se puede realizar desde node.Js o desde VSC.

Algunos comandos que se estarán utilizando con frecuencia:

- **Instalacion** : npm install -g create-react-app
- **Nuevo proyecto**: create-react-app pruebasReact
- **Pruebas**: npm start
- **compilar**: npm run build

El diseño web con React.js **se basa totalmente en jerarquías de componentes, que son preferiblemente pequeños fragmentos reutilizables en que podemos dividir una interfaz**. No hay ninguna imposición sobre cuántos debemos crear o como de reutilizables deben ser, pero esto es clave a la hora de crear un código elegante y profesional.

Al programar con React usamos frecuentemente sintaxis de **ECMAScript6**. Además de las novedades del estándar deberás también conocer el **lenguaje de marcas JSX, que es prácticamente HTML5 pero con ciertas diferencias** (como convertir todos los atributos a un formato camelCase), y lo usaremos para crear la “vista” de nuestros componentes.

Tipos de componente

Los componentes pueden ser de dos tipos:

- **Con estado**: Se dice que un componente tiene estado cuando usa una variable de tipo objeto donde almacena **información sobre su situación actual** (que varía en el tiempo). Por ejemplo: un componente que sea una caja de texto podría tener en su estado la longitud de caracteres del texto que contiene.
- **Sin estado**: Son aquellos que no necesitan tener su propio estado. Ya que o bien son completamente estáticos o reciben las variables que necesitan como propiedades desde su componente padre.

Si queremos tener el menor número de componentes con estado posible, ya que así el código será más fácil de mantener. Es mejor tener componentes principales con estado que pasen como parámetro/propiedades todo lo necesario a sus hijos.

Por lo dicho anteriormente se determinan dos posibles opciones para crear un componente:

- **Componente funcional**: Este método solo sirve para crear componentes sin estado. Y básicamente se trata de una función que recoge unas propiedades/parámetros (opcionalmente) y devuelve su código JSX, así de simple.

```
function Welcome(props) {
return <h1>Hello, {props.name}</h1>;
}
```

- **Componente basado en clase**: Usando la sintaxis ES6 para definir clases se hereda de Component y se implementa un constructor donde recoge las propiedades y un método render, que debe retornar el código JSX que visualiza el componente. Si queremos disponer de estado o sobrescribir los métodos de su ciclo de vida, entonces tenemos que crear un componente basado en clase.

```
import React,{Component} from 'react';
class Welcome extends React.Component {
constructor(props){
super(props);
this.state = { //un ejemplo de cómo se define el estado desde el constructor.
op1: "prueba",
op2: 3
};
render() {
return <h1>Hello, {this.props.name}</h1>;
}
}
```

Otro dato básico: cuando estemos escribiendo JSX podemos incluir en él variables javascript o llamadas a funciones que devuelvan JSX usando las llaves, como vemos con {this.props.name}

Jerarquía

La jerarquía de componentes se va creando, incluyendo unos componentes (hijo) dentro de otros (padres), y esto se hace de forma muy sencilla indicando el componente a insertar como un **tag html** que puede además recibir parámetros en forma de atributos. Estos tag se incluyen

directamente en el código JSX de los componentes, para incluir unos dentro de otros e ir componiendo la interfaz.

```
function Welcome(props) {
return <h1>Hello, {props.name}</h1>;
}
function App() {
return (
<div>
<Welcome name="Sara" />
<Welcome name="Cahal" />
<Welcome name="Edite" />
</div>
);
}
```

Evidentemente deberá existir un primer padre o raíz a partir del cual vamos enganchando al resto de componentes. Esto se logra con la función **ReactDOM.render**, que como primer parámetro indica el componente raíz y como segundo el elemento del documento **html** donde vamos a **montar** toda la aplicación.

```
function App() {
return (
<div>
<Welcome name="Sara" />
<Welcome name="Cahal" />
<Welcome name="Edite" />
</div>
);
}
ReactDOM.render(
<App />,
document.getElementById('root')
);
```

Flujo de propiedades y eventos

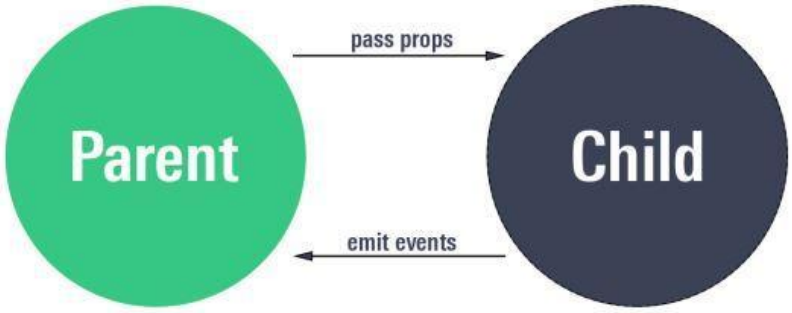
Otro fundamento de React es el modo en que se comparte información entre los diferentes componentes. Acostumbrados a tener **listeners** en cualquier lugar en **javascript** tradicional, ahora se usan las propiedades para pasar información de padres a hijos, y los eventos se disparan y van subiendo hacía el padre donde en algún momento serán capturados y "manejados".

Las props se pasan de padres a hijos y los eventos se disparan de hijos a padres.

En el siguiente ejemplo vemos como el elemento hijo es un botón que recibe el texto que muestra como propiedad desde su padre, y por el contrario, cuando dispara el evento **onClick** (nota el camelCase de JSX), el evento viaja hasta el padre, en concreto hasta la función manejadora que también pasó por propiedades.

```
//PADRE
handleOnClick(e){
this.setState({
loading: true
});
}
render(){
return(
<ChildComponent text="Click on me!" handleOnClick={this.handleOnClick} />
)
}

//HIJO
function ChildComponent(props){
return(<button onClick={this.props.handleOnClick}>{this.props.text}</Button>);
}
```



Ejemplo práctico:

1. realizar la instalación de React: **npm install -g create-react-app**
2. Crear nuestro primer Proyecto: **create-react-app pruebasReact**
3. Después de crear el proyecto se deben modificar algunos archivos para poder comenzar a trabajar, abrimos el archivo **src/App.css** y eliminamos el código y copiamos lo siguiente:

```
.App {
  text-align: center;
}

.App-logo {
```

```
  height: 40vmin;
  pointer-events: none;
}

@media (prefers-reduced-motion: no-preference) {
  .App-logo {
    animation: App-logo-spin infinite 20s linear;
  }
}

.App-content {
  background-color: #61dafb;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}

form input {
  height: 26px;
  border-radius: 5%;
  display: flex;
  vertical-align: auto;
}

form button {
  cursor: pointer;
  display: inline-block;
  text-align: center;
  text-decoration: none;
  margin: 2px 0;
  border: solid 1px transparent;
  border-radius: 4px;
  padding: 0.5em 1em;
  color: #ffffff;
  background-color: darkgreen;
```

```
  height: 30px;
  width: 120px;
}

h3 {
  margin: 0;
}

.list {
  display: flex;
  margin: 5px;
  flex-direction: row;
  align-items: flex-end;
}

.btn-delete {
  cursor: pointer;
  display: inline-block;
  text-align: center;
  text-decoration: none;
  border: solid 1px transparent;
  border-radius: 4px;
  color: #ffffff;
  background-color: red;
  height: 30px;
  width: 30px;
}
```

4. ahora en **src/App.js** eliminamos el contenido y copiamos lo siguiente:

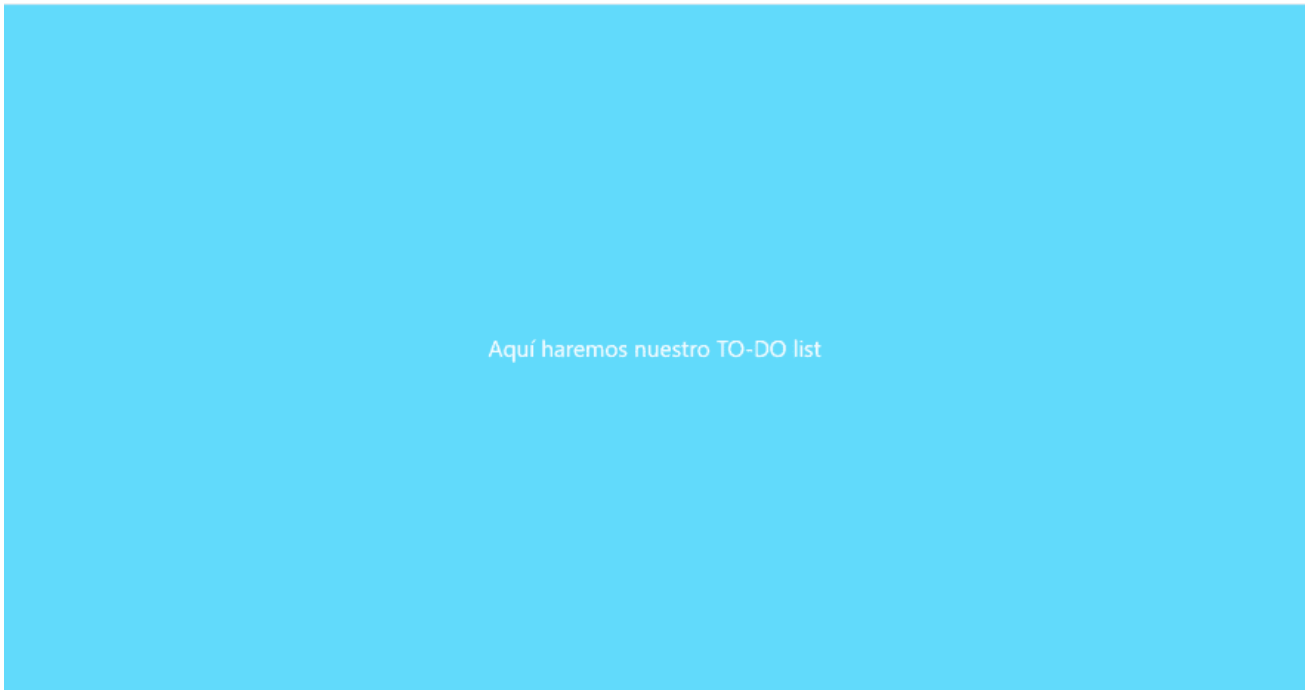
```
import React from 'react';
import './App.css';


const  App = () => {

  return (
    <div className="App">
      <div className="App-content">
        <p>
          Aquí haremos nuestro TO-DO list
        </p>

      </div>
    </div>
  );
}
export default App;
```

5. Probamos los cambios: **npm start**



6. Es todo, no necesitamos nada más con lo que a estilos corresponde, vamos a empezar con la parte **jsx**, lo primero que tenemos que hacer es crear una carpeta en la raíz de **src** a la cual llamaremos **components**, ahí estarán nuestros componentes, para este ejemplo solo crearemos 2, vamos al primero, dentro de components creamos un archivo llamado **Todo.jsx**, con React podemos utilizar la extensión **.js o .jsx** para nuestros componentes sin ningún problema agregamos el siguiente código dentro de nuestro componente **Todo**:

```
import React from 'react'

const Todo = () => {
    return (
        <h1>Todo component</h1>
    )
}
export default Todo
```

7. Ahora también dentro de la carpeta **components** creamos un archivo llamado **Form.jsx** con el siguiente código:

```
import React from 'react'

const Form = () => {
    return (
        <h1>Form component</h1>
    )
}
export default Form
```

8. Ahora que tenemos ambos componentes hagamos lo siguiente dentro del componente **Form**, agregamos lo siguiente:

```
import React from 'react';
import Todo from '../components/Todo';

const Form = () => {
    return (
        <>
          <h1>Form component</h1>
        <Todo></Todo>
        </>
    )
}
export default Form
```

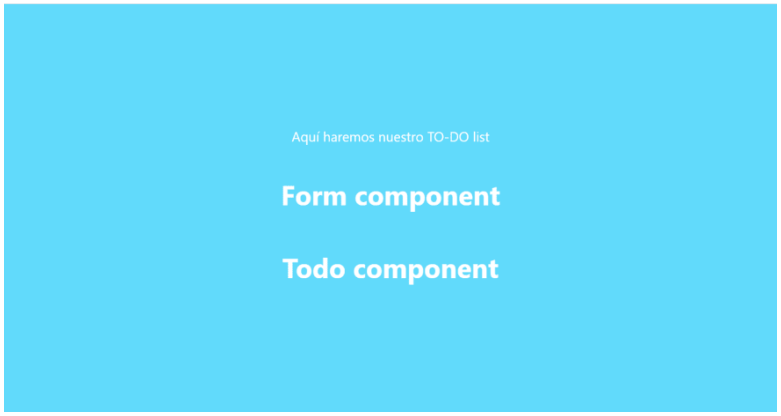
9. Ahora en nuestro componente **App.js** hacemos lo siguiente:

```
import React from 'react';
import './App.css';
import Form from './components/Form';

const  App = () => {

  return (
    <div className="App">
      <div className="App-content">
        <p>
          Aquí haremos nuestro TO-DO list
        </p>
        <Form />
      </div>
    </div>
  );
}
export default App;
```

10. Hacer pruebas: **npm start** , ya podemos ver que nuestros dos componentes se han incluido correctamente:



Construyendo la lógica de nuestra aplicación

11. Hasta este punto tenemos todo lo que necesitamos para trabajar, la mayor parte del trabajo lo vamos a realizar en nuestro componente **Form** así que vamos a él, estamos construyendo una aplicación con tareas, entonces necesitamos las tareas, vamos al código:

```
import React,{useState} from 'react';
import Todo from '../components/Todo';

const Form = () => {
    const [todos, setTodos] = useState([
        {todo: 'todo 1'},
        {todo: 'todo 2'},
        {todo: 'todo 3'}
])

    return (
        <>
        <h1>Form component</h1>
      <Todo></Todo>
      </>
    )
}
export default Form
```

Lo que hicimos aquí fue importar **useState** y declarar un estado dentro de nuestro componente **Form**, el cual es un array de objetos que serán nuestras tareas.

¿Debo usar var o const o let? última versión de Javascript **ecmascript6**.

- **var**: define un contexto de funciones.
- **let**: define un contexto de bloque (funciones, condicionales, iteraciones)
- **const**: define un contexto de bloque y el valor de la variable es inmutable.

Presentando Hooks

Hooks son una nueva característica en **React 16.8**. Estos te permiten usar el estado y otras características de React sin escribir una clase.

¿Qué es un Hook? Un Hook es una función especial que permite "conectarse" a características de React. Por ejemplo, **useState** es un Hook que te permite añadir el estado de React a un componente funcional.

¿Cuándo debería usar un Hook? Si creas un componente funcional y descubres que necesitas añadirle estado, antes había que crear una clase. Ahora puedes usar un **Hook** dentro de un componente funcional existente.

12. En nuestro componente **Todo** realizamos los siguientes cambios:

```
import React from 'react'

const Todo = ({todo}) => {
    return (
        <>
            <h3>{todo}</h3>
        </>
    )
}

export default Todo
```

Eliminamos la etiqueta h1 y en su lugar agregamos un h3 el cual se encarga de imprimir un **prop** que es únicamente el nombre de nuestra tarea.

13. Vamos de nuevo al componente **Form** y agregamos lo siguiente:

```
import React,{useState} from 'react';
import Todo from '../components/Todo';

const Form = () => {
    const [todos, setTodos] = useState([
        {todo: 'todo 1'},
        {todo: 'todo 2'},
        {todo: 'todo 3'}
])

    return (
        <>
        {
            todos.map((value, index) => (<Todo todo={value.todo} />))
        }
      </>
    )
}
export default Form
```

Muy bien, lo que hacemos es recorrer nuestro array de tareas (**todos**) el cual habíamos inicializado con tres objetos, dentro de la función **map** incluimos nuestro componente Todo y le pasamos el **prop** que necesita, si vamos al navegador tenemos el siguiente resultado:

¡Perfecto! ya tenemos nuestras tareas renderizadas, pero ahora necesitamos empezar a agregar más tareas.

14. Vamos al código, a realizar unas modificaciones a nuestro componente **Form** para que se vea así:

```
import React,{useState} from 'react';
import Todo from '../components/Todo';

const Form = () => {
    const [todo, setTodo] = useState({})
    const [todos, setTodos] = useState([
        {todo: 'todo 1'},
        {todo: 'todo 2'},
        {todo: 'todo 3'}
])
const handleChange = e => setTodo({[e.target.name]: e.target.value})
const handleClick = e => console.log('click click')

    return (
        <>
        <form onSubmit={e => e.preventDefault()}>
            <label>Agregar tarea</label><br />
            <input type="text" name="todo" onChange={handleChange}/>
            <button onClick={handleClick}>agregar</button>
        </form>
        {
            todos.map((value, index) => (
                <Todo todo={value.todo} />
            ))
        }

    </>
    )
}
export default Form
```

Vamos a empezar con la nueva constante que definimos arriba de nuestro estado de TAREAS, agregamos este fragmento de código.

const [todo, setTodo] = useState({}),

El cual inicializa un nuevo estado que nos va a servir para agregar UNA tarea, ya que el estado anterior, nos ayuda a crear UNA LISTA de TAREAS,

Vamos a la siguiente, función:

const handleChange = e => setTodo({[e.target.name]: e.target.value}),

captura el evento **change** de nuestro **input** el cual se mira ahora así

<input type="text" name="todo" onChange={handleChange}/>

El input tiene un atributo **name** en el cual el nombre es el mismo que la **key** de nuestros objetos (tareas), esto es porque en la función **handleChange** recibimos como parámetro el evento como tal y esta representado con la **variable e**, nos permite acceder a algunas propiedades entre ellas

al **name del input y al value del mismo** y estos se encuentran dentro de **target**, es por eso que hacemos **setTodo({[e.target.name]: e.target.value})** dentro de **handleChange**.
Modificamos el estado de nuestra aplicación para capturar una nueva tarea, la siguiente función se llama **handleClick**, su única función en este momento es imprimir en consola un mensaje, pero más adelante eso cambiará.

const handleClick = e => console.log('click click')

Para terminar con los detalles nuestro formulario también ejecuta un evento, solo que esta vez no creamos una nueva función sino que ejecutamos una **arrow function** directamente, de nuevo capturamos el evento y hacemos un **e.preventDefault(),** es para que al hacer **submit** de nuestro formulario no se refresque nuestro navegador.
Bien por último nuestro botón ejecuta en el evento **onClick** la función **handleClick**, si en efecto esta que solo imprime un mensaje por el momento.

15. Vamos a nuestro componente **Todo** para realizar unas modificaciones y se vea de la siguiente manera:

```
import React from 'react'

const Todo = ({todo, index, deleteTodo}) => {
    return (
        <>
            <div className="list">
                <h3>{todo}</h3> <button className="btn-delete" onClick={() => deleteTodo(index)}>x</button>
            </div>
        </>
    )
}

export default Todo
```

Agregamos un **boton** con una clase **btn-delete**, **OJO** en react no podemos usar la palabra **class** en nuestro código jsx, recuerda que jsx NO **ES HTML** sino una extensión del lenguaje JavaScript y class es una palabra reservada del lenguaje, en su lugar usamos **className**.
Continuemos, agregamos dos **props (propiedades)** más **index y deleteTodo**, estas dos propiedades la cual una es un entero y la otra es una función nos ayudarán más adelante para eliminar tareas, la función **deleteTodo** se ejecuta en el evento **onClick** del botón que acabamos de agregar (comunicación hijos a padres), hemos terminado nuestro componente **Todo** aquí ya no haremos más.

16. Volvamos a nuestro componente **Form** modificamos **handleclick** y de igual manera agregamos un nuevo método, así es **deleteTodo** que será el **prop** que le pasaremos a nuestro componente Todo al igual que **index**, entonces el código se vería de la siguiente manera.

```
import React,{useState} from 'react';
import Todo from '../components/Todo';

const Form = () => {
    const [todo, setTodo] = useState({})
    const [todos, setTodos] = useState([
        {todo: 'todo 1'},
        {todo: 'todo 2'},
        {todo: 'todo 3'}
])
const handleChange = e => setTodo({[e.target.name]: e.target.value})
const handleClick = e => {
    if(Object.keys(todo).length === 0 || todo.todo.trim() === '') {
        alert('el campo no puede estar vacio')
        return
    }
    setTodos([...todos, todo])
}

const deleteTodo = indice => {
    const newTodos = [...todos]
    newTodos.splice(indice, 1)
    setTodos(newTodos)
}

    return (
        <>
        <form onSubmit={e => e.preventDefault()}>
            <label>Agregar tarea</label><br />
```

17. La función **handleClick** ahora ya tiene una funcionalidad, lo primero que hacemos es validar que nuestro **input** no este vacío y después solo lo agregamos al estado, hacemos uso del **dentro de setTodos** para agregar nuestra nueva tarea y conservar todas las que tenemos.

El **spread operator,** se puede usar para tomar una matriz existente y agregarle otro elemento mientras se conserva la matriz original.

- var colors = ['red', 'green', 'blue'];
- var refColors = [...colors, 'yellow'];
- //colors => ['red', 'green', 'blue']
- //refColors => ['red', 'green', 'blue', 'yellow']

La función **deleteTodo** obtiene todas las tareas, y elimina la tarea con el **indice** que le pasamos como parámetro y por último actualizamos el estado con la nueva lista de tareas.

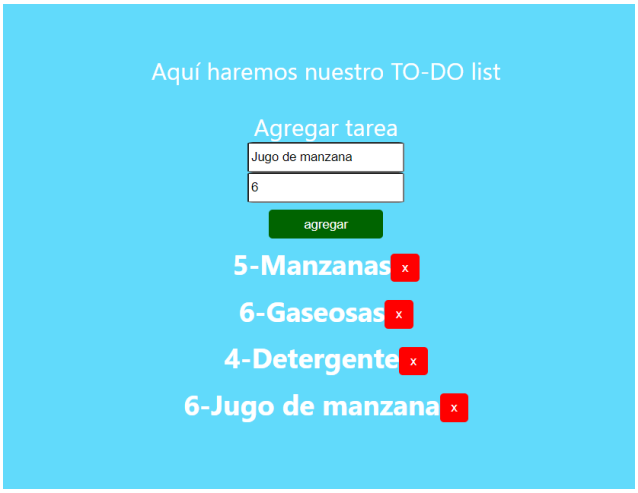
18. para terminar con nuestro ejemplo vamos a modificar **Form.jsx** la función **map** dentro del **return** para pasarle a **Todo.jsx** los **props (propiedades)** que necesita:

```
todos.map((value, index) => (
  <Todo todo={value.todo} key={index} index={index} deleteTodo={deleteTodo}/>
))
```

19. Hacer pruebas: **npm start ,** nuestra aplicación se ve así:

V. DISCUSION DE RESULTADOS

En base al ejemplo de la guia crear una web app para realizar una lista de compra para el supermercado, la cual podría verse de la siguiente manera:



VII. BIBLIOGRAFIA

- React, una biblioteca de JavaScript. (2013-2020). Facebook, Inc. Jordan Walke. Recuperado de https://es.reactjs.org/docs/getting-started.html