

# 1-Fundamentos de las herramientas de desarrollo

---



Veremos herramientas utiles para el desarrollo con **REACT** como **NPM**, **BABEL**, **WEBPACK** aunque tambien se utilizaran

## Entorno de desarrollo de REACT JS

Se puede implementar **REACT** utilizando la libreria sin compilar pero lo mas recomendado es utilizar **JSX** el cual es un lenguaje compilado. Para la implementacion de **REACT** se utilizara el modulo *create-react-app*.

## Explicacion de NPM, Webpack y babel

**NPM**: Es un gestor de modulos de JS.

**BABEL**: Es un traductor de codigo JS con estandares nuevos, lo que permite que el proyecto sea utilizable en cualquier navegador, aunque este no soporte los ultimos estandares del **ecmascript**.

**WEBPACK**: Permite ulizar modulos para reutilizar codigos, es muy util utilizarlo junto con **BABEL**.

## Chrome DevTools

Para abrir las **DevTools** se utiliza *ctrl+shift+i*.

Las **DevTools** posee una *consola* la cual permite una interaccion entre el programa y el navegador.

## Creando Stack con REACT

Para crear una app de **REACT** utilizamos el comando **CREATE-REACT-APP nombre app**, este comando puede ser instalado con **NPM**.

## Comandos utiles

- **NPM START**: Ejecuta la aplicacion en un servidor local.
- **NPM RUN**:

## Estructura del proyecto

Dentro de la carpeta **public** se encuentra el **INDEX.html** en el cual se puede encontrar un *div* de clase *root* que en donde se generara nuestra aplicacion. El archivo **manifest.json** se encuentra las cosas necesarias para realizar una **PWA**.

Dentro de la carpeta **src** se encuentran todos los archivos de la aplicacion , en el **Index.js** se encuentra importa el modulo *serviceWorker* junto con el archivo **serviceWorker.js** se utiliza para generar una **PWA**.

## 2-Introduccion REACT CORE

---

Esta seccion permite comprender el entorno de **REACT**, y todo su entorno como **JSX** y las **expresiones** las cuales permiten generar codigo *html* de forma de reutilizar codigo. Junto con la aplicacion de codigo de **CSS** para mejor las paginas web.

Los temas vistos en esta seccion son:

1. Introduccion a JSX
2. Componentes
3. JSX en profundidad
4. EcmaScript 6
5. Manejo de Eventos
6. Extras

## Introduccion a JSX

**JSX** es una combinacion entre **JS** y **HTML** la cual permite indexar directamente **HTML** en codigo **JS**.

Creacion de elementos e insercion en el DOM

Utilizar **REACT** debo importar el modulo **React** y **ReactDOM**.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

const app = <h1>Hola React</h1>;

ReactDOM.render(app, document.getElementById('root') );
// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

## Expresiones

Para insertar una expresion en **JSX**, utilizando llaves, dentro de ellos puedo colocar cualquier expresion valida de **JS**.

```
function crearFrase( name ){
  return 'Bienvenido ' + name
}

let name = 'Lautaro';

const app = <h1> { crearFrase( name ) } </h1>
```

## Componentes

Los componentes permiten la reutilización de código. Existen 2 grandes grupos de componentes: los **FUNCIONALES** y **BASADOS EN CLASES**. Los más recomendables para utilizar son los componentes **FUNCIONALES** ya que necesitan menor cantidad de líneas de código gracias a los **HOOKS**.

### Funcionales

Son funciones que retornan una única etiqueta de **HTML**. Para llamar un componente funcional se debe utilizar la nomenclatura clásica de **HTML**.

```
react-fundamentos > src > JS index.js > ...
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  function TarjetaFruta(){
8      return(
9          <div>
10             <h1> Lista de Frutas </h1>
11          </div>
12      );
13  }
14
15
16  ReactDOM.render(<TarjetaFruta />, document.getElementById('root') );
17  // If you want your app to work offline and load faster, you can change
18  // unregister() to register() below. Note this comes with some pitfalls.
19  // Learn more about service workers: https://bit.ly/CRA-PWA
20  serviceWorker.unregister();
```

### Props

Las **props** son variables de entrada de los componentes, los componentes que se le pasan a un **props** se deben utilizar llaves aunque para los datos de tipo **String** son opcionales.

```
react-fundamentos > src > .js index.js > TarjetaFruta
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  function TarjetaFruta(props){
8    return(
9      <div>
10        <h1> Lista de Frutas </h1>
11        <ul>
12          <li> { props.name } </li>
13        </ul>
14      </div>
15    );
16  }
17
18
19  ReactDOM.render(<TarjetaFruta name='sandia' />, document.getElementById('root') );
20  // If you want your app to work offline and load faster, you can change
21  // unregister() to register() below. Note this comes with some pitfalls.
22  // Learn more about service workers: https://bit.ly/CRA-PWA
23  serviceWorker.unregister();
```

## Clases

Son clases que extienden de **React.Component** y debe implementar un metodo **render** el cual renderiza el componente.

Los componentes basados en clases soportan una actualizacion dinamica, utilizando una variable de clase **state** y para actualizar el componente se utiliza el metodo *setState*.

**ACLARACION:** Para los metodos que utilicen **this** es recomendable utilizar el metodo **bind( this )** para todos los metodos que lo requieran. Es recomendable utilizar inicializadores de propiedades.

## Inicializadores de Propiedades

Se crea un atributo fuera del constructor y permite prescindir del mismo ya que la unica llamada necesaria es la del constructor del padre.

```
class TarjetaFruta extends React.Component{
  state = {
    cantidad: 0,
  }

  agregar = () => {
    this.setState( {
      cantidad: this.state.cantidad + 1
    } );
  }

  quitar = () => {
    const cantidadNueva = (this.state.cantidad>0) ? this.state.cantidad-1 : 0;
    this.setState({
      cantidad: cantidadNueva
    });
  }

  render(){
    return(
      <div>
        <h3>Fruta: { this.props.name }</h3>
        <h4>Cantidad: {this.state.cantidad }</h4>
        <p>Value: ${ this.props.price }</p>
        <button onClick={ this.agregar }>+1</button>
        <button onClick={ this.quitar }>-1</button>
      </div>
    );
  }
}
```

## Factorizacion de componentes

Es muy recomendable tener los componentes separados en diferentes carpetas, para lograr tener el código mucho más ordenado.

## JSX en profundidad

Los componentes deben comenzar con mayúscula, ya que sino **Babel** no tomara como válido.

Para agregar una clase a una etiqueta se utiliza **className**.

**JSX** permite declarar módulos de componentes, es decir, es posible crear objetos **JSON** donde cada atributo sea un componente.

## CSS en React

Para indexar código **CSS** debo colocar el atributo **style** y pasar como argumento un objeto que contenga las propiedades.

```
class TarjetaFruta extends React.Component{
  state = { ...
}



Si el estilo posee un guion medio (-) se utiliza la notación camelCase. JSX permite la implementación de operadores ternarios dentro de los objetos, lo cual hace mucho más sencilla la implementación de estilos según los estados.



6 / 14


```

```
render(){
  const hasItems = this.state.cantidad > 0;

  const divStyle = {
    border: '1px solid black',
    width: '150px',
    height: '300px',
    borderRadius: '0.5em',
    padding: '1em',
    marginBottom: '1em',
    marginLeft: '2em',
    background: hasItems ? 'linear-gradient(45deg, #adebad, #33cc33)' : 'linear-gradient(45deg, black, red)',
    color: hasItems ? 'white' : 'black',
    transition: 'all 450ms ease-out',
  }

  return(
    <div style={
      divStyle
    } >
      <h3>Fruta: { this.props.name }</h3>
      <h4>Cantidad: {this.state.cantidad }</h4>
      <p>Value: ${ this.props.price }</p>
      <p>
        Total: ${ this.props.price * this.state.cantidad }
      </p>
      <button onClick={ this.agregar }>+1</button>
      <button onClick={ this.quitar }>-1</button>
    </div>
  );
}
```

## CSS utilizando clases

Es recomendable utilizar dentro de cada carpeta del componente sus estilos, lo que permite tener componentes aun mas modularizados. **JSX** permite importar archivos **CSS** directamente, esto sucede gracias al paquete de **Webpack**.

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

import './style.css';

class TarjetaFruta extends React.Component {
  state = { ...
}

  agregar = () => { ...
}

  quitar = () => { ...
}

  render() {
    const hasItems = this.state.cantidad > 0;
    const clases = `box ${ hasItems ? 'box-activa' : '' }`;
    return (
      <div className = { clases } >
        <h3>Fruta: { this.props.name }</h3>
        <h4>Cantidad: {this.state.cantidad }</h4>
        <p>Value: ${ this.props.price }</p>
        <p>
          Total: ${ this.props.price * this.state.cantidad }
        </p>
        <button onClick={ this.agregar }>+1</button>
        <button onClick={ this.quitar }>-1</button>
      </div>
    );
  }
}
```

## Sistemas de Modulos

Cuando se desea utilizar un sistema de modulos de **CSS** el arvhivo debe ser nombrado de la siguiente forma:

***nombre.modules.css***

## Ecmascript 6

Object.assing() 👍



Este metodo sirve para combinar objetos, pero si existe una coincidencia entre los atributos de los objetos estas seran sobrescritas por la ultima encontrada. La solucion de este problema puede es utilizar el metodo **Assing** sobre los atributos que se pisan creando un objetonuevo dentro de ese atributo.

## Operador Spread [ ... ]

Cuando poseo objetos dentro de objetos se utiliza el operador **Spread**. El cual remplaza al metodo **Assing** con una sintaxis mucho mas amigable.

```
const resultado = {  
  ...region,  
  ...perfil,  
  ...social,  
  regiones: {  
    ...region.regiones,  
    ...social.regiones  
  }  
}
```

EL operador **spread** sirve para concatenar **arrays** de una forma muy util

## Pasando props con Spread

Podemos utilizar el operador **Spread** para pasar parametros a los componetes de la forma:

```
return (  
  <div className="App">  
    <Gato name='Garfield' age='2 years' {...datos} />  
    <Clicks/>  
  </div>  
)
```

## Mutacion de componenetes utilizando funciones

Debido a que el metodo **setState** es asincrono es muy util utilizar una funcion de callback dentro del metodo.

```
add = () => {  
  this.setState( (state, props) => {  
    {  
      video: {  
        ...state.video,  
        likes: state.video.likes + 1  
      }  
    }  
  })  
}
```

## Manejo de eventos

### Eventos de Mouse

Existen varios tipos de eventos de *Mouse* utiles cuando se desea utilizar **REACT**, algunos de ellos son:

- onClick
- onMouseDown
- onMouseUp
- onMouseMove

Para mayor informacion se recomienda ir a la pagina oficial de **REACT**.

### Eventos tipo Input

Las etiquetas de **Input** poseen eventos muy utiles como:

- onChange
- onPaste
- onCopy

Los cuales ayudan a manejar la web de forma dinamica.

### Crear eventos personalizados en REACT

**REACT** permite crear eventos personalizados, lo que hasta ahora lo hemos visto como **props** aunque ahora podremos pasar informacion de un componente padre a un componente hijo y viceversa.

En el componenete padre utilizamos:

```
<Hijo onSaluda={ manejador }/>
```

Mientras que en el componente hijo se utiliza

```
class Hijo extends React.Component{  
  render(){  
    return(  
      <div className={ styles.box }>  
        <h2>  
          Hola  
        </h2>  
  
        <button onClick={ ()=>{  
          this.props.onSaluda( 'HOLA' )  
        } }>  
          CLICK ME!  
        </button>  
      </div>  
    )  
  }  
}
```

## Extras

### Inyectando HTML en componentes

Si bien esta practica esta desaconsejada, en algunas ocasiones puede ser util esto se logra utilizando el atributo ***dangerouslySetInnerHTML*** y pasandole un **JSON** con el atributo **\_\_html**.

### Prop especial children

Si se desea utilizar etiquetas dentro de elementos componentes se puede utilizar la **prop children**, para ello es necesatrio abrir y cerrar la etiqueta del componente.

```

const Subtitle = (props) =>{
  const subStyles = {
    color: props.uiColor
  }

  return(
    <h1 style={ subStyles }>
      { props.children }
    </h1>
  )
}

class Title extends React.Component{

  state = {
    uiColor: '#2a231'
  }

  render(){
    return(
      <div className={ styles.title }>
        <Subtitle uiColor={ this.state.uiColor }>
          Super <em>Lauti</em>
        </Subtitle>
      </div>
    )
  }
}

```

## Destructuración avanzada

La destructuración nos permite que el código sea mucho más legible, y menos confuso.

```

const saluda = (user) =>{
  const { name, country } = user;

  console.log(
    `Hola soy ${name} y vivo en ${country}`
  );
}

```

Es válido utilizar valores por defecto y destructuración concatenada

```
const saluda = (user) =>{
  const { name, social: { tw: 'Tw de la empresa' } } = user;

  console.log(
    `Hola soy ${name} y mi twitter es ${ tw }`
  );
}
```

### Elementos sin etiquetas con *fragment*

El **fragment** es una etiqueta de **REACT** que permite omitir la envoltura de etiquetas en los componentes, las formas de utilizar esta etiqueta son:

```
const Computacion = () =>(  
  <React.Fragment>  
    <li>Monitor</li>  
    <li>Mouse</li>  
    <li>Teclado</li>  
  </React.Fragment>  
)  
  
const Ropa = () =>(  
  <>  
    <li>Camisa</li>  
    <li>Pantalon</li>  
    <li>Remera</li>  
  </>  
)
```

### Portals

Los portals permiten renderizar etiquetas en otra etiqueta que no se **root**, lo cual es muy util a la hora de hacer modals.

```
class PortalModal extends React.Component{  
  
  render(){  
    return ReactDOM.createPortal((  
      <div>  
        <h1> { this.props.children } </h1>  
      </div>  
    ), document.getElementById( 'modal-root' ))  
  }  
}
```

Para utilizar el modal debo llamar el componenete en mi **App.js**.

**ACLARACION: SE DEBE CREAR UN DIV CON EL ID *modal-root*, pues REACT no es magico.**

## Control de datos de entrada

Para utilizar el control de datos es necesario utilizar el modulo **PropTypes** de **JS**, el cual se puede instalar con **NPM** mediante el comando **npm i prop-types**.

Cuando se desea utilizar el control de datos, se debe implementar de la forma:

```
Profile.proptype = {  
  name: PropTypes.string.isRequired,  
  biografia: PropTypes.string,  
  email: PropTypes.string  
}
```

Donde en este caso **Profile** es un componente basado en clases, pero la nomenclatura para componentes **funcionales** es la misma.

## Datos de entrada por defecto

Como es posible la comprobacion de datos, tambien se puede definir valores por defecto a las **props**, implementacion es identica a la de las **proptypes**, con la unica diferencia que debo llamar el atributo **defaultProps**

# 3-Iterando Listas

---

En este apartado se tratara el tema de **iteradores**, ya sea sobre **arrays**, **JSON arrays** y **JSON**.

- 1.