

# ReactJS

## Instalación y Configuración

### Presentación:

En esta clase veremos qué es React Js, cómo funciona, qué ventajas tiene respecto de otras tecnologías.

Luego comenzaremos a realizar la puesta en marcha de nuestro primer desarrollo, como instalarlo y como configurarlo.

### Objetivos:

#### Que los participantes:

- Comprendan que es React JS y que ventajas nos brinda
- Aprendan a instalar y configurar React JS
- Realicen una primer aplicación en React JS

### Bloques temáticos:

- Virtual DOM
- Ventajas de utilizar React JS
- Node JS
- Crear una aplicación utilizando el CLI
- Componentes
- Tipos de componentes
- JSX
- Componentes con clases
- Componentes con funciones
- Estructura de directorio

## Virtual DOM

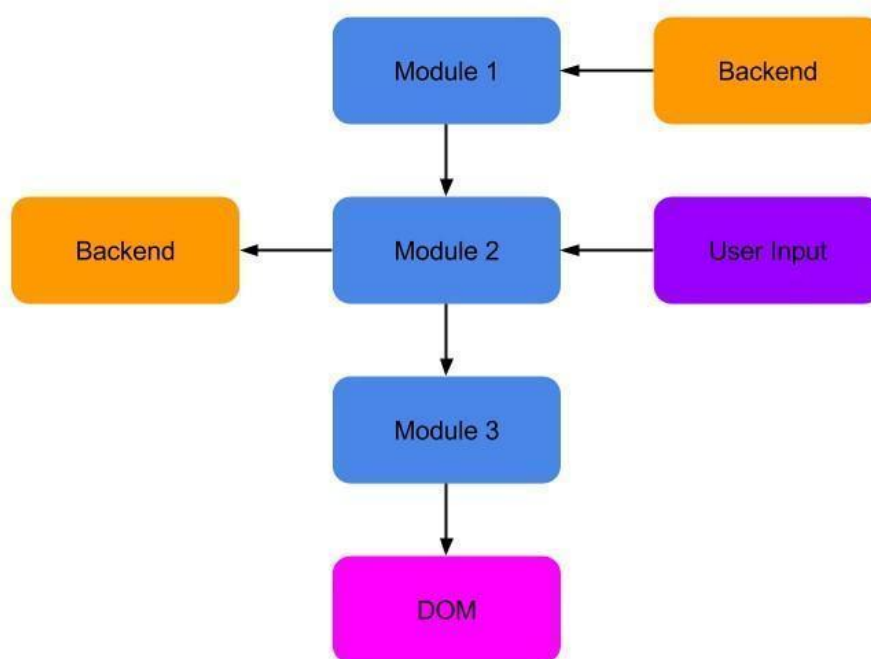
### La clave de React JS

El secreto de ReactJS para tener un performance muy alto, es que implementa algo llamado Virtual DOM y en vez de renderizar todo el DOM en cada cambio, que es lo que normalmente se hace, este hace los cambios en una copia en memoria y después usa un algoritmo para comparar las propiedades de la copia en memoria con las de la versión del DOM y así aplicar cambios exclusivamente en las partes que varían.

Esto puede sonar como mucho trabajo, pero en la práctica es mucho más eficiente que el método tradicional pues si tenemos una lista de dos mil elementos en la interfaz y ocurren diez cambios, es más eficiente aplicar diez cambios, ubicar los componentes que tuvieron un cambio en sus propiedades y renderizar estos diez elementos, que aplicar diez cambios y renderizar dos mil elementos.

Son más pasos a planear y programar, pero ofrece una mejor experiencia de usuario y una planeación muy lineal.

Una característica importante de ReactJS es que promueve el flujo de datos en un solo sentido, en lugar del flujo bidireccional típico en Frameworks modernos, esto hace más fácil la planeación y detección de errores en aplicaciones complejas, en las que el flujo de información puede llegar a ser muy complejo, dando lugar a errores difíciles de ubicar.



## ¿Cómo funciona el virtual DOM?

JavaScript puro es el lenguaje declarativo utilizado para crear aplicaciones web interactivas, React es una biblioteca de JavaScript que ayuda a los desarrolladores a construir interfaces de usuario interactivas y reutilizables.

Imagina que tienes un objeto que es un modelo en torno a una persona. Tienes todas las propiedades relevantes de una persona que podría tener, y refleja el estado actual de la persona. Esto es básicamente lo que React hace con el DOM.

Ahora piensa, si tomamos ese objeto y le hacemos algunos cambios. Se ha añadido un bigote, unos bíceps y otros cambios. En React, cuando aplicamos estos cambios, dos cosas ocurren:

- En primer lugar, React ejecuta un algoritmo de “diffing”, que identifica lo que ha cambiado.
- El segundo paso es la reconciliación, donde se actualiza el DOM con los resultados de diff.

Lo que hace React, ante estos cambios, en lugar de tomar a la persona real y reconstruirla desde cero, sólo cambiaría la cara y los brazos. Esto significa que si usted tenía el texto en una entrada y una actualización se llevó a cabo, siempre y cuando el nodo padre de la entrada no estaba programado para la actualización, el texto se quedaría sin ser cambiado.

## Ventajas de utilizar React JS

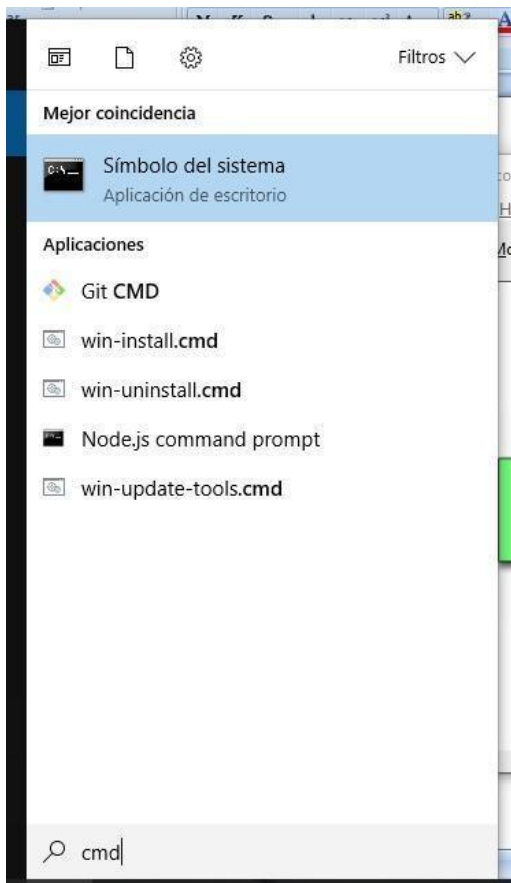
- Performance con DOM Virtual
- Flujo unidireccional de datos y eventos a través de una jerarquía de componentes modulares con punto único de entrada (programación reactiva)
- Binding unidireccional entre el view (vista) y modelo, se evita complejidad y explosión de eventos difíciles para debugging.
- Se usa realmente por sus creadores
- Se integra directamente con implementaciones del paradigma también reactivo de gestión de datos Flux (como Redux) que maneja los datos de la aplicación en una división también jerárquica de componentes contenedores vinculados al estado de los datos en el store, y que pasan estos datos y funciones gestores de esos datos a componentes puros que demuestran los datos en forma predecible sin efectos secundarios y ofrecen puntos de interacción al usuario.
- Es fácil incluir rendering (proyección del componente en el DOM virtual como nodos de elementos con estilo y comportamiento, o sea HTML + CSS +

JavaScript) en el lado del servidor con funciones sencillas, para lograr las llamadas aplicaciones universales (antes se llamaban isomórficas).

## Crear una aplicación utilizando el CLI

### Trabajar con la consola de windows

#### Abrir la consola



#### Ubicarnos en un directorio específico

Con el comando `cd` podemos ingresar al directorio sobre el cual vamos a crear nuestra aplicación en react. Con `cd..` Volvemos al directorio anterior.

```
G:\sites>cd react
```

Luego estamos dentro del directorio react

```
G:\sites\react>
```

Para ver más sobre el uso de consola en Windows:

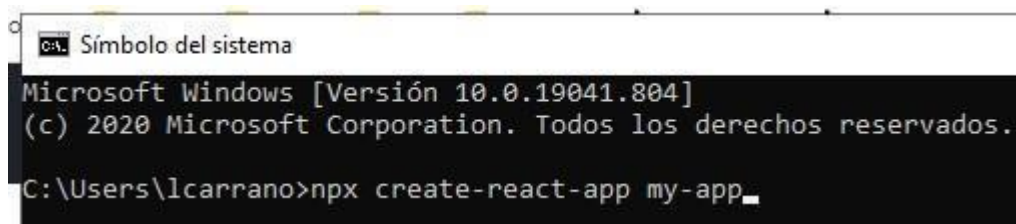
<https://computerhoy.com/paso-a-paso/software/comandos-esenciales-consola-windowscmd-que-debes-conocer-77943>

Para ver más sobre el uso de la consola en Linux / mac:

<https://swcarpentry.github.io/shell-novice-es/02-filedir/index.html>

## Crear una aplicación

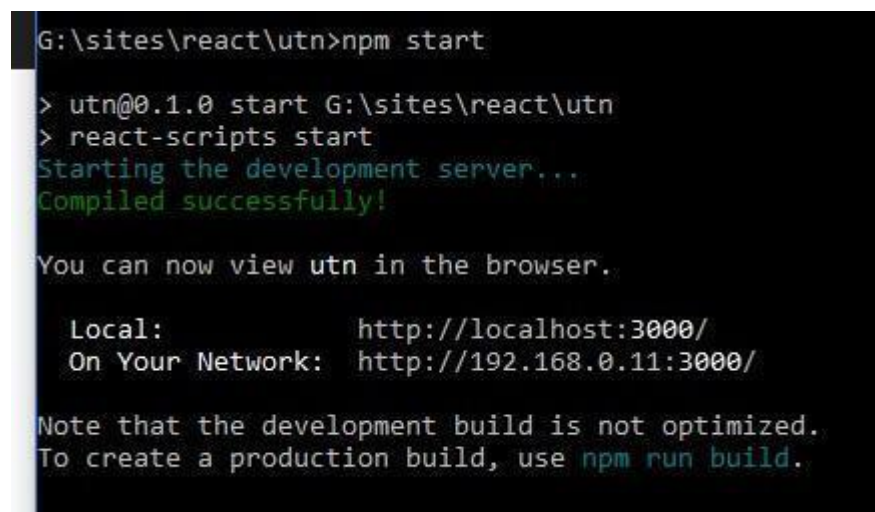
Para crear nuestra aplicación debemos ejecutar `npx create-react-app my-app`



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19041.804]
(c) 2020 Microsoft Corporation. Todos los derechos reservados.
C:\Users\lcarrano>npx create-react-app my-app_
```

## Ejecutar aplicación en el navegador

Para ejecutar una aplicación y poder acceder desde el navegador debemos ejecutar (dentro del directorio de la aplicación creada) `npm start`



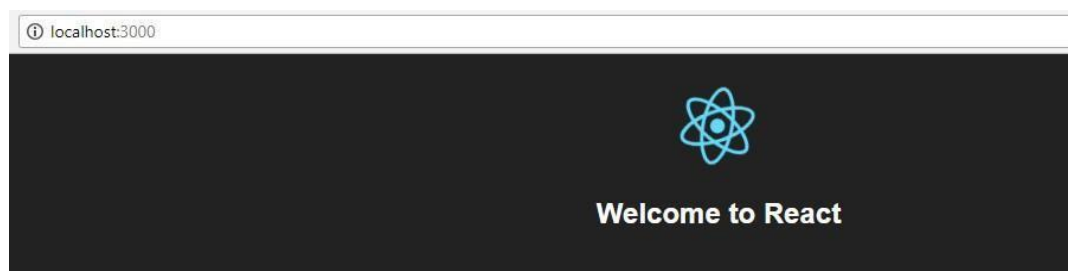
```
G:\sites\react\utn>npm start
> utn@0.1.0 start G:\sites\react\utn
> react-scripts start
Starting the development server...
Compiled successfully!

You can now view utn in the browser.

Local:      http://localhost:3000/
On Your Network:  http://192.168.0.11:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

El resultado obtenido será el siguiente:



## Componentes

### En pocas palabras

Comenzamos un tema importante en React, ya que en esta librería realizamos un desarrollo basado en componentes. Básicamente quiere decir que, mediante anidación de componentes podremos crear aplicaciones completas de una manera modular, de fácil mantenimiento a pesar de poder ser complejas.

En React JS existen dos categorías recomendadas para los componentes: los componentes de presentación y los componentes contenedores.

Los **componentes de presentación** son aquellos que simplemente se limitan a mostrar datos y tienen poca o nula lógica asociada a manipulación del estado, es preferible que la mayoría de los componentes de una aplicación sean de este tipo porque son más fáciles de entender y analizar.

Los **componentes contenedores** tienen como propósito encapsular a otros componentes y proporcionarles las propiedades que necesitan, además se encargan de modificar el estado de la aplicación por ejemplo usando Flux o Redux para despachar alguna acción y que el usuario vea el cambio en los datos.

### Ventajas del enfoque

- Favorece la separación de responsabilidades, cada componente debe tener una única tarea.
- Al tener la lógica de estado y los elementos visuales por separado es más fácil reutilizar los componentes.
- Se simplifica la tarea de hacer pruebas unitarias.
- Puede mejorar el rendimiento de la aplicación.
- La aplicación es más fácil de entender.

### Composición de componentes

Así como en programación funcional (paradigma de programación que se basa en la idea de que los programas deben ser escritos en términos de funciones matemáticas puras) se pasan funciones como parámetros para resolver problemas más complejos, creando lo que se conoce como composición funcional, en ReactJS podemos aplicar este mismo patrón mediante la composición de componentes

Las aplicaciones se realizan con la composición de varios componentes. Estos componentes encapsulan un comportamiento, una vista y un estado. Pueden ser muy complejos, pero es algo de lo que no necesitamos preocuparnos cuando estamos desarrollando la aplicación, porque el comportamiento queda dentro del componente y no necesitamos complicarnos por él una vez se ha realizado.

En resumen, al desarrollar crearemos componentes para resolver pequeños problemas, que por ser pequeños son más fáciles de resolver y en adelante son más fáciles de visualizar y comprender. Luego, unos componentes se apoyarán en otros para resolver problemas mayores y al final la aplicación será un conjunto de componentes que trabajan entre sí. Este modelo de trabajo tiene varias ventajas, como la facilidad de mantenimiento, depuración, escalabilidad, etc.

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

En este caso vemos como el componente `app` inyecta al componente `welcome`

## Tipos de componentes

### Características de los componentes de presentación

- Orientados al aspecto visual
- No tienen dependencia con fuentes de datos (e.g. Flux)
- Reciben callbacks por medio de props
- Pueden ser descritos como componentes funcionales.
- Normalmente no tienen estado

Ejemplo de componente de presentación

```
// Componentes de presentación
class Item extends React.Component {
  render() {
    return (
      <li>
        <a href='#'>{this.props.valor}</a>
      </li>
    );
  }
}

class Input extends React.Component {
  render() {
    return (
      <input type='text' placeholder={this.props.placeholder} />
    );
  }
}

class Titulo extends React.Component {
  render() {
    return (
      <h1>{this.props.nombre}</h1>
    );
  }
}
```



En este fragmento de código definimos algunos componentes de presentación (Header, Item e Input) que son mostrados en la página dentro de un componente contenedor.

Los componentes de presentación no tienen estado y sólo contienen código para mostrar información, y para hacer el código más simple dichos componentes se crean con una función en vez de una clase. Además reciben de su componente padre las propiedades necesarias.

También es posible escribir estos mismos componentes de forma funcional, es decir, en vez de que sean definidos con la palabra `class`, serán creados como simples funciones que regresan el contenido que se mostrará:

```
// Componentes de presentación de forma funcional
const Titulo = ({ nombre } = props) => (
  <h1>{nombre}</h1>;
const Item = (props) => (
  <li><a href='#'>{props.valor}</a></li>;
const Input = (props) => (
  <input type='text' placeholder={props.placeholder} />;
```

Usando esta sintaxis las propiedades se reciben como parámetros de la función e incluso es posible aplicar destructuring para obtener las variables que nos interesan por separado.

## Características de los componentes contenedores

- Orientados al funcionamiento de la aplicación
- Contienen componentes de presentación y/o otros contenedores
- Se comunican con las fuentes de datos
- Usualmente tienen estado para representar el cambio en los datos

```
// Contenedor
class AppContainer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      temas: ['JavaScript', 'React JS', 'Componentes']
    };
  }
```

```
}  
render() {  
  const items = this.state.temas.map(t => (<Item valor={t} />));  
  return (  
    <div>  
      <Titulo nombre='List Items' />  
      <ul>{items}</ul>  
      <Titulo nombre='Inputs' />  
      <div>  
        <Input placeholder='Nombre' /><br />  
        <Input placeholder='Apellido' />  
      </div>  
    </div>  
  );  
}
```

Por otra parte el componente contenedor define los datos contenidos en la aplicación y también los manipula, después crea los componentes hijos y los muestra en el método render.

No hay una diferencia sintáctica entre ambos componentes, la misma es 100% conceptual

## JSX

JSX es una extensión de JavaScript creada por Facebook para el uso con su librería React. Sirve de preprocesador (como Sass o Stylus a CSS) y transforma el código a JavaScript.

Te puede parecer que estás mezclando código HTML dentro de tus ficheros JavaScript, pero nada más lejos de la realidad. A continuación te lo explico. JSX puede recordarte a un lenguaje de plantillas, pero viene con todo el poder de JavaScript.

React al basar el desarrollo de apps en componentes, necesitamos crear elementos HTML que definen nuestro componente, por ejemplo <div>, <p>, <img>.

## ¿Por qué usar JSX?

React acepta el hecho de que la lógica de renderizado está intrínsecamente unida a la lógica de la interfaz de usuario: cómo se manejan los eventos, cómo cambia el estado con el tiempo y cómo se preparan los datos para su visualización.

En lugar de separar artificialmente tecnologías poniendo el maquetado y la lógica en archivos separados, React separa intereses con unidades ligeramente acopladas llamadas “componentes” que contienen ambas.

React no requiere usar JSX, pero la mayoría de la gente lo encuentra útil como ayuda visual cuando trabajan con interfaz de usuario dentro del código Javascript. Esto también permite que React muestre mensajes de error o advertencia más útiles.

Puedes utilizar comillas para especificar strings literales como atributos:

```
const element = <div tabIndex="0"></div>;
```

También puedes usar llaves para insertar una expresión JavaScript en un atributo:

```
const element = <img src={user.avatarUrl}></img>;
```

No pongas comillas rodeando llaves cuando insertes una expresión JavaScript en un atributo. Debes utilizar comillas (para los valores de los strings) o llaves (para las expresiones), pero no ambas en el mismo atributo

Dado que JSX es más cercano a JavaScript que a HTML, React DOM usa la convención de nomenclatura **camelCase** en vez de nombres de atributos HTML.

Por ejemplo, **class** se vuelve **className** en JSX, y **tabindex** se vuelve **tabIndex**.

## Ejemplo

Crear un componente sin utilizar JSX sería algo como esto

```
var image = React.createElement('img', {  
  src: 'react-icon.png',  
  className: 'icon-image'  
});  
var container = React.createElement('div', {
```

```
    className: 'icon-container'  
  }, image);  
var icon = React.createElement('Icon', {  
  className: 'avatarContainer'  
}, container);  
ReactDOM.render(  
  icon,  
  document.getElementById('app'));
```

Obteniendo el siguiente resultado:

```
<div class='icon-container'  
  <img src='icon-react.png' class='icon-image' />  
</div>
```

```
.icon-image {  
  width: 100px  
}  
.icon-container {  
  background-color: #222;  
  width: 100px  
}
```

```
var Icon = (  
  <div className='icon-container'  
    <img  
      src='icon-react.png'  
      className='icon-image'  
    />  
  </div>  
)  
ReactDOM.render (Icon, document.getElementById('app'))
```

Ver mas en <https://es.reactjs.org/docs/jsx-in-depth.html>

## Componentes con clases

Son componentes declarados como clases. Al declarar estos componentes los mismos deben heredar de la clase `Component`

```
import React, { Component } from 'react';
class Contacto extends Component {
  constructor(props) {
    super(props)
    console.log(this.props)
  }
  render() {
    let prueba = "dsadas"
    return (
      <div className="App">
        {this.props.data.map(d => <div>{d}</div>)}
        Contacto
      </div>;
    )
  }
}
export default Contacto;
```

## Render

Todo componente de clase en React, tiene un método Render que es el que se encarga de renderizar en el navegador el HTML correspondiente al componente.

Este método se llama automáticamente cuando se crea un componente y cuando el estado del componente se actualiza.

En este método es donde usamos JSX para facilitar el desarrollo y creación de elementos HTML. Veamos un ejemplo:

```
import React from 'react'
class MyComponent extends React.Component {
  constructor() {
    super()
  }
  render() {
    return (
      <div>
        <span>Hola!, soy un componente</span>
      </div>
    )
  }
}
```

React DOM compara el elemento y sus hijos con el elemento anterior, y solo aplica las actualizaciones del DOM que son necesarias para que el DOM esté en el estado deseado.

## Componentes con funciones

Son componentes de react declarados como funciones, por defecto el componente app viene declarado como un componente funcional

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Home from './Home'
import Contacto from './Contacto'
import Menu from './Menu'
function App() {
  return (
    <div className="App">
      Hola Mundo
    </div>);
}
export default App;
```

En el caso de las funciones las mismas no disponen de un método o función para el renderizado. Se renderiza lo que se encuentre declarado dentro del return

## Estructura de directorio

Si observamos el código de la aplicación creada veremos lo siguiente en el archivo **index.js**:

```
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();
```

Acá podemos observar que estamos renderizando el componente `<App />` sobre el elemento con id `root` del DOM. ¿Que tiene el componente `<App />`?

Para responder a esta pregunta debemos ir al archivo **App.js**

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';
class App extends Component {
  render() {
    var app = <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <h1 className="App-title">Welcome to React</h1>
      </header>
      <p className="App-intro">
        To get started, edit <code>src/App.js</code> and save to reload.
      </p>
    </div>;
    return (app);
  }
}
export default App;
```

Acá podemos observar como el componente **App** renderiza una imagen de un párrafo.

También podemos ver como el src de la imagen lo obtiene utilizando el **import** de javascript y no definiendo su ruta absoluta y/o relativa en el código.

Volviendo al archivo **index.js**, ¿en donde está el elemento del DOM con id **root**?

Veamos el archivo **index.html**

```
<html><body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.
```



```
You can add webfonts, meta tags, or analytics to this file.  
The build step will place the bundled scripts into the <body> tag.  
To begin the development, run 'npm start' or 'yarn start'.  
To create a production bundle, use 'npm run build' or 'yarn build'
```

```
-->
```

```
</body>
```

```
</html>
```

Este será el html que se renderiza en nuestro navegador. Luego utilizando react se renderiza el contenido del componente **App** sobre el elemento con id **root**

En caso de querer modificar los metas de nuestra página debemos hacerlo sobre el archivo **index.html**

## Bibliografía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt.

<https://reactjs.org/tutorial/tutorial.html>

<https://reactjs.org/docs/hello-world.html>

<https://carlosvillu.com/introduccion-a-reactjs/>

<https://platzi.com/blog/react-js-de-javascript/>

<http://code.ezakto.com/react/introduccion-a-react.html>

<https://es.reactjs.org/docs/jsx-in-depth.html>