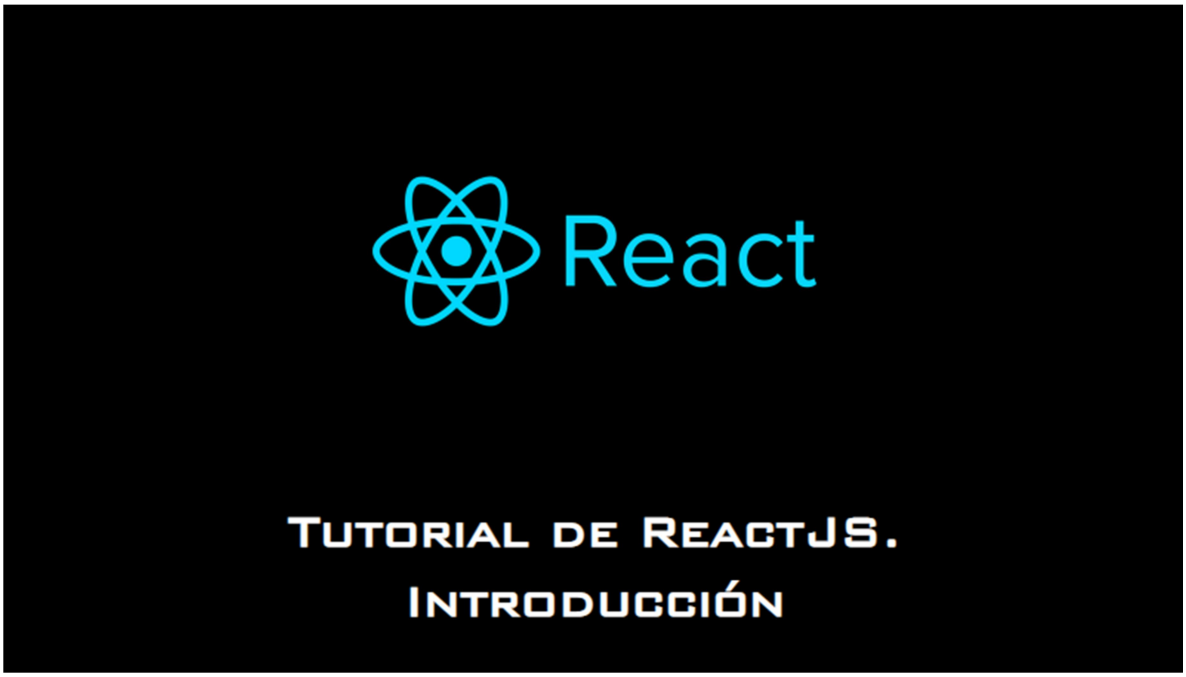

React
Tutorial de ReactJS.
Introducción

Hoy toca escribir sobre otra de las tecnologías que uso en varios de los proyectos en los que trabajo, **ReactJS**. Se que existe muchísima información en la red sobre esta librería, aunque en muchos casos inconexa.

## ¿Qué es React.js?

Como muchos ya sabréis, **ReactJS** es una **librería Javascript** desarrollada por **Facebook** y diseñada para ayudarnos a crear **SPA's** (Single Page Application), su objetivo concretamente es tratar de facilitar la tarea de **desarrollar interfaces** de usuario. Podríamos decir que React es la **V** en un contexto en el que se use el patrón MVC o MVVM.

Hace uso del paradigma denominado programación orientada a componentes. Dichos componentes se representan como clases que heredan de la clase `Component` cuyo único requerimiento especial es especificar el método render que define cuál será el contenido del mismo:

```
class MyComponent extends React.Component {
  render() {
    return (
      <h1>Hello World</h1>
    );
  }
}
```

La definición de dichos componentes se realiza usando una sintaxis especial llamada **JSX**que permite escribir etiquetas HTML dentro de JavaScript para mejorar la expresividad del código. Usar JSX no es obligatorio, pero si es muy recomendable. Para más información sobre JSX puedes consultar la documentación oficial.

# Configuración del entorno mediante "Create React App"

Normalmente cuando vamos a construir una aplicación Web con Javascript tendremos que lidiar con una cantidad de ingente de herramientas como gestores de paquetes, transpiladores, linkers, builders, etc. El equipo de desarrollo de Facebook ha sabido ver esta problemática y se ha sacado de la manga el proyecto Create React App, el cual realizará por nosotros toda la configuración inicial necesaria para poder empezar a desarrollar con React.

**Create React App** se puede utilizar con el nuevo gestor de dependencias Yarn, creado también por la gente de Facebook, o con el clásico **NPM**. En el artículo haré uso de NPM, aunque te aconsejo que le des una oportunidad a Yarn, tiene muy buena pinta.

El único requisito imprescindible para poder hacer uso de Create React App con NPM es tener instalado en el sistema una versión de NodeJs >= 4. Si ya dispones de npm puedes instalar `create react app` como cualquier otro paquete:

```
npm install -g create-react-app
```

Una vez instalado puedes inicializar el proyecto:

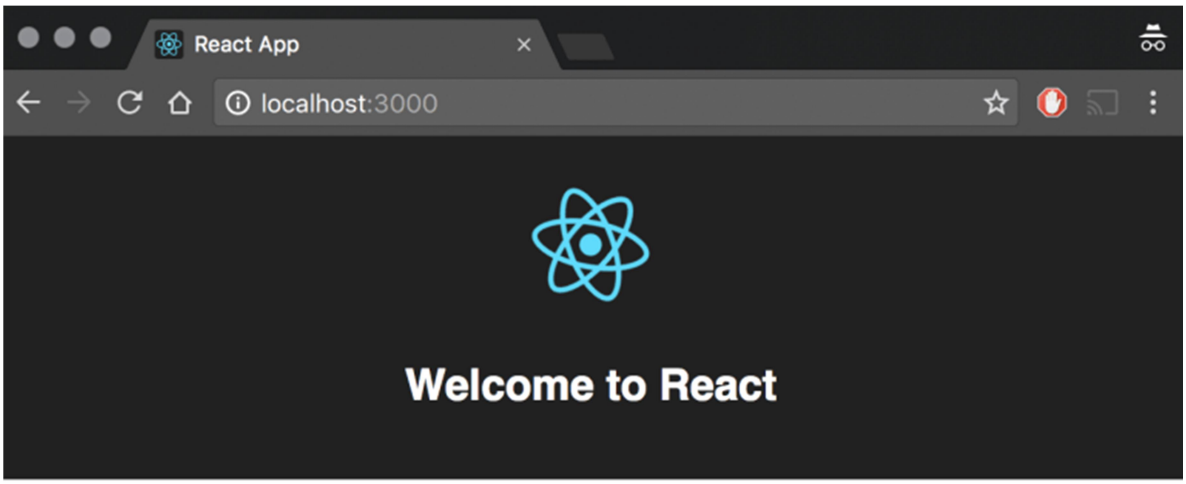
```
create-react-app MyWebApp
```

Con este simple gesto tendrás configurado JavaScript ES6 con React, Webpack, Babel y Eslint, nada de instalar dependencias, ni de crear tareas. Está todo listo para ejecutar el servidor de desarrollo, y probar la aplicación:

```
cd MyWebApp

npm start
```

Con el servidor corriendo, dirígete a la url 127.0.0.1:3000 para ver la aplicación en funcionamiento:




El proyecto generado tendrá una estructura tal que así:

```
MyWebApp/
  README.md
  node_modules/
  package.json
  public/
    index.html
    favicon.ico
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
```

- **node_modules**: contiene las dependencias npm del proyecto
- **public**: esta es la raíz de nuestro servidor donde se podrá encontrar el index.html, el archivo principal y el favicon.
- **src**: es el directorio principal donde vamos a colocar los archivos de nuestros componentes.

Además encontrarás varios archivos sueltos, un readme, el .gitignore y el **package.json**, este último contiene las dependencias de npm además de la información del proyecto.

Es importante tener en cuenta que para que Create React App funcione correctamente tenemos que tener obligatoriamente el fichero principal de html en "public/index.html" y punto de entrada de javascript en "scr/index.js".

## El componente principal y el punto de entrada javascript

Como he comentado el punto de entrada a la aplicación es el fichero scr/index.js, en este se inicializa el componente principal App.js, a través del método `ReactDOM.Render`. Dicho método recibe como primer parámetro el componente a renderizar y como segundo el elemento del DOM donde el componente va ser renderizado:

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


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

A continuación vemos el código auto-generado que corresponde al componente principal:

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Bienvenido a React</h2>
        </div>
          <p className="App-intro">
            Lista de usuarios
          </p>
      </div>
    );
  }
}

export default App;
```

Antes de continuar desarrollando el ejemplo voy eliminar unas cuantas líneas de este componente para dejarlo, por ahora, lo más simple posible:

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


class App extends Component {
  render() {
    return (
      <h1>Hello World!</h1>
    );
  }
}
```

```
export default App;
```

En la primera línea `import React…` se está importando la librería React y la clase Component de la cual van a heredar todos los componentes que se creen mediante clases. Éstas requieren del método `render()` para poder funcionar.

En la versión previa de Javascript se utilizaba la función `React.createClass` para inicializar componentes, gracias a ES6 y a su azúcar sintáctico, esto se ha simplificado.

## Propiedades (props) de react

Las propiedades de un componente (props) pueden definirse como los atributos de configuración para dicho componente. Éstas son recibidas desde un nivel superior, normalmente al realizar la instanciación del componente y por definición son inmutables.

Siguiendo con el ejemplo, voy a implementar el componente `User`, el cual contiene dos props `name` y `user`, las cuales redenrizará en un elemento de lista `li`.

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

class User extends Component {
  render () {
    return (
      <li>
        {this.props.name} - {this.props.email}
      </li>
    );
  }
}

export default User;
```

Las props, básicamente, son el mecanismo principal de React para pasar datos de un componente padre a un componente hijo.

### Anidación de componentes

Una vez creado el componente User, definiremos el componente `UserList`, cuyo objetivo será renderizar una lista de componentes `User`:

```
class UserList extends Component {
  render () {
    return (
        <ul>
          {this.props.users.map(u => {
            return (
              <User
                key={u.id}
                name={u.name}
                email={u.email}
              />
            );
          })}
        </ul>
    );
  }
}
```

En el código del componente anterior renderizará una lista de usuarios, para ello hace uso del método `map`, con el cual itera sobre cada uno de los elementos del array de usuarios que contiene la propiedad `this.props.users`, esta prop será recibida desde el componente `App`.

Map devuelve por cada elemento un componente `User`, el cual recibe vía props el nombre, el email y una key. La propiedad key es un identificador que usa React en las listas, para renderizar los componentes de forma más eficiente.

## Estado en los componentes

Podría definirse el estado de un componente como una representación del mismo en un momento concreto, algo así como una instantánea del componente. Dicho estado se iniciará con un valor por defecto.

Existen dos tipos de componentes con y sin estado, también denominados statefull y stateless, respectivamente.

## Componentes stateless

Todos los componentes implementados hasta el momento han sido stateless, sin estado. Este tipo de componentes podrían representarse como una función pura:

```
    function User(props) {
    return (
        <li>{props.name} - {props.email}</li>
    );
}
```

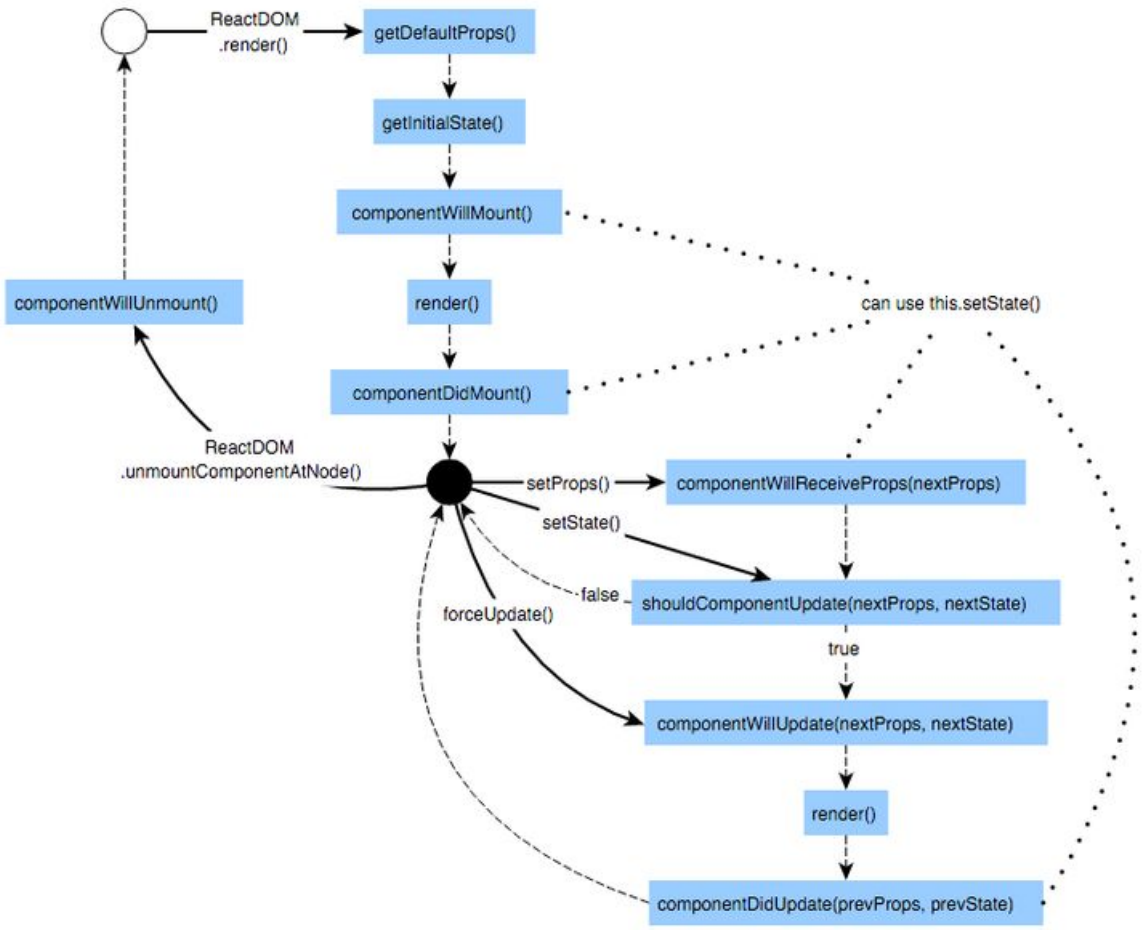
Las funciones puras por definición no tienen efectos colaterales, con lo cual este tipo de componentes no admite ciclos de vida. Para no complicar las cosas, continuaremos creando nuestros componentes como clases.

## Componentes statefull

Los componentes con estado permiten mantener datos propios a lo largo del tiempo e implementar comportamientos en sus diferentes métodos del ciclo de vida.

## El ciclo de vida de un componente en React

El ciclo de vida no es más que una serie de estados por los cuales pasa todo componente a lo largo de su existencia, básicamente se pueden clasificar en tres etapas de montaje o inicialización, actualización y destrucción. Dichas etapas tienen correspondencia en diversos métodos, que nosotros podemos implementar para realizar acciones concretas cuando estos van sucediendo. Aunque en el ejemplo que estamos desarrollando no voy a hacer uso del ciclo de vida, haré un pequeño inciso para describir sus métodos sin profundizar excesivamente:




**Métodos de inicialización:**

- `componentWillMount()`: se ejecuta antes de que se renderice el componente por primera vez, es muy util para manejar ciertos datos necesarios para la representación del componente o declarar ciertos eventos. Las referencias a los elementos del componente aun no están disponibles.
- `componentDidMount()`: se dispara justo después del primer renderizado, es decir el DOM ya está disponible. Este es el sitio adecuado para realizar peticiones AJAX, setIntervals o integrar librerías de terceros.

**Métodos de actualización:**

- `componentWillReceiveProps(nextProps)`: es ejecutado cuando las propiedades se van a actualizar, recibe el próximo valor que va a tener el objeto de propiedades.

- `shouldComponentUpdate(nextProps, nextState)`: se lanza antes del render y decide si nuestro componente se re-renderiza o no. Recibe dos parametros, las nuevas propiedades y el nuevo estado.
- `componentWillUpdate(nextProps, nextState)`: se ejecutará justo después de que shouldComponentUpdate devuelva true, está pensado para preparar al componente para su actualización por lo que se debe evitar modificar estados en este punto.
- `componentDidUpdate(prevProps, prevState)`: se invoca justo después de haberse producido la actualización del componente, los cambios ya están trasladados al DOM.

**Métodos de desmontaje:**

- `componentWillUnmount()`: es el único método que interviene en el desmontaje de un componente, es invocado justo antes de que el componente se desmonte, es ideal para realizar operaciones de limpieza como listeners de eventos o temporizadores.

Continuando con el ejemplo, voy a modificar el componente App para asignarle un estado inicial que almacene un array con varios objetos "user". Para ello sobreescribiremos el método constructor del componente asignando al estado inicial (`this.state`) el array de usuarios.

Finalmente, en el método render renderizará un componente del tipo `UserList`, al cual se le pasa el estado a través de la prop `users`. Quedando el componente tal que así:

```
class App extends Component {
  constructor() {
    super();
    this.state = {
      users: [
        {id: 1, name: "miguel", email: "test@miguelgomez.io"},
        {id: 2, name: "test", email: "test@test.es"}
      ]
    };
  }

  render() {
```

```
    return (
        <UserList users={this.state.users} />
    );
  }
}
```

Gracias al estado se pueden añadir nuevos usuarios al array los cuales se renderizarán automáticamente. Esto es posible ya que, como hemos visto en los ciclos de vida, el estado tiene la particularidad de que cuando cambia el método render vuelve a ejecutarse.

## Propagación de eventos

Los eventos, al contrario que las propiedades, se propagan de hijos a padres. Es decir, son lanzados por los componentes hijos y el padre es el encargado de gestionarlos.

Veamos esto en nuestro ejemplo, para ello vamos añadir al componente UserForm un formulario con dos campos, uno para el email y otro para el nombre.

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

export default class UserForm extends Component{
  render(){
    return (
      <form onSubmit={this.props.onAddUser}>
          <input type="text" placeholder="Nombre" name="name" />
          <input type="email" placeholder="Email" name="email" />
          <input type="submit" value="Guardar" />
      </form>
    );
  }
}
```

Cada vez que se pulse el botón guardar el formulario disparará el evento `onSubmit()`, el cual llama a una función que recibirá del componente padre a través de la propiedad `props.OnAddUser` (a esto se le conoce como **Callback**).

La función callback se define en el componente App, `handleOnAddUser(event)` y será la encargado de manejar el evento.

```
  class App extends Component {
constructor() {
  super();
  this.state = {
    users: [
      {id: 1, name: "miguel", email: "miguelghz@miguelgomez.io"},
      {id: 2, name: "test", email: "test@test.es"}
    ]
  };
}

handleOnAddUser (event) {
  event.preventDefault();
  let user = {
    name: event.target.name.value,
    email: event.target.email.value
  };
  this.setState({
    users: this.state.users.concat([user])
  });
}

render() {
  return (
      <div>
        <UserList users={this.state.users}  />
        <UserForm onAddUser={this.handleOnAddUser.bind(this)} />
```

```
        </div>
    );
  }
}
```

El método `handleOnAddUser` recibe como parámetro un objecto event, el cual contiene toda la información del evento, tanto su comportamiento como los valores de los inputs del formulario. Para evitar que el evento dispare su comportamiento por defecto (en este caso recargar la página), ejecutaremos el método `preventDefault()` antes de capturar los valores de los campos "name" y "user".

Por último, actualizaremos el array, para ello en lugar de modificarlo añadiendo el nuevo elemento con el método `push`, usamos el método `concat`. De esta manera se creará un nuevo array en lugar de modificar el existente, manteniendo así cierta inmutabilidad en el estado del componente.