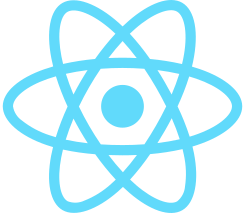
**Curso Practico de React JS.**

**¿Qué es React JS?**



**React** es una librería desarrollada por Facebook que nos ayuda a construir interfaces de usuario interactivas para todo tipo de aplicaciones: web, móviles o de escritorio.

Cada pequeña parte de nuestra página web la conoceremos como “**Componente**”. Cada componente se encargará de una función en específico. Además, podremos reutilizar nuestros componentes siempre que los necesitemos.

Al unir todos nuestros componentes tendremos una página web que nos permite cambiar, actualizar o eliminar elementos de forma muy sencilla.

# DOM, Virtual DOM y React DOM

El **DOM** es el código HTML que se transforma en páginas web.

Cada vez que cambiamos alguna parte del DOM, también estamos actualizando el HTML con el que interactúan nuestros usuarios. El problema es que todas las operaciones, comparaciones y actualizaciones en el DOM son muy costosas.

El **Virtual DOM** es una herramienta que usan tecnologías como React y Vue para mejorar el rendimiento (performance) y velocidad de nuestras aplicaciones.

Es una copia exacta del DOM, pero mucho más ligera, ya que los cambios no actualizan el verdadero HTML de nuestras páginas web. Gracias al Virtual DOM podemos hacer operaciones y comparaciones de forma sumamente rápida.

Recuerda que los cambios en el Virtual DOM no afectan el HTML que ven los usuarios, así que debemos estar sincronizando constantemente las copias con el DOM. Pero no te preocupes, **React DOM** lo hace por nosotros.

# Create React App y Tipos de Componentes

### **Inicialización de un proyecto en React**

Creación de nuestro sitio web usando la plantilla por defecto de [create-react-app](https://facebook.github.io/create-react-app/docs/getting-started):



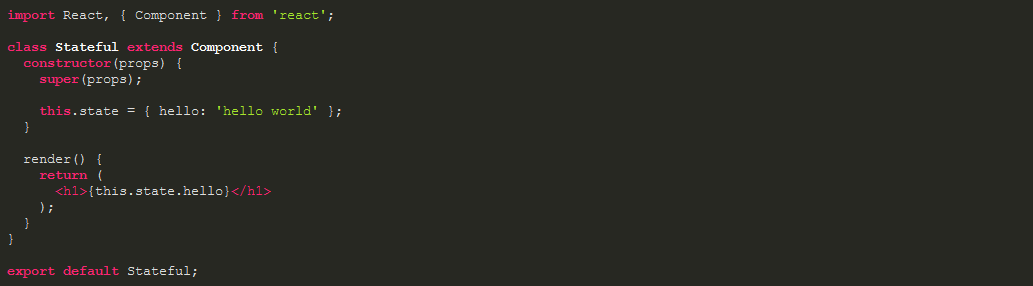
Iniciar el servidor de desarrollo:



### **Creación y Tipos de Componentes**

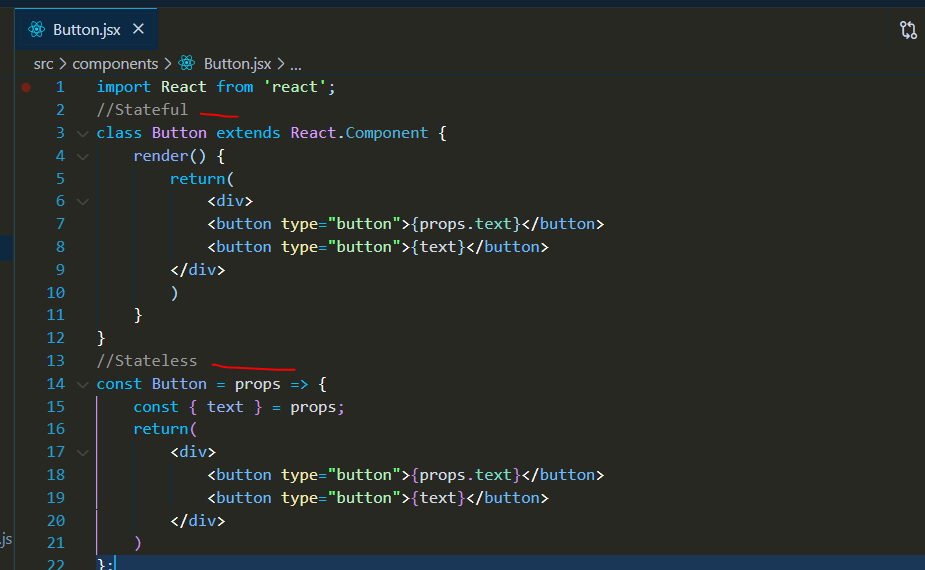
Los nombres de nuestros componentes deben empezar con una letra mayúscula, al igual que cada nueva palabra del componente. Esto lo conocemos como Pascal Case o Upper Camel Case.

Los componentes **Stateful** son los más robustos de React. Los usamos creando clases que extiendan de **React.Component**. Nos permiten manejar estado y ciclo de vida (más adelante los estudiaremos a profundidad).

También tenemos componentes **Stateless** o **Presentacionales**. Los usamos creando funciones que devuelvan código en formato JSX (del cual hablaremos en la próxima clase).



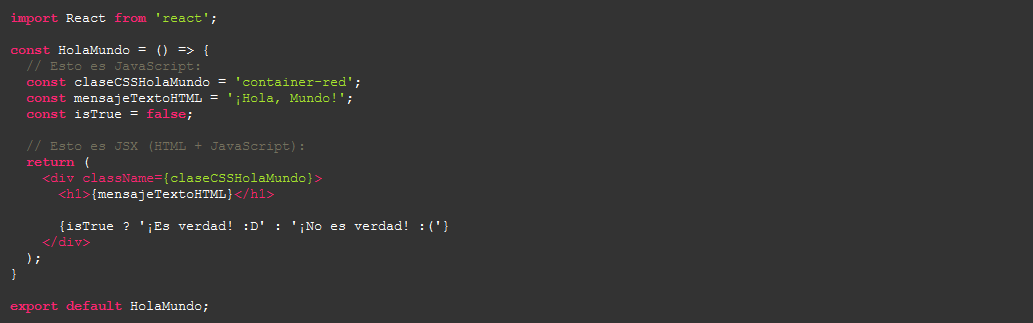
**Stateless VS Stateful.**



# JSX: JavaScript + HTML

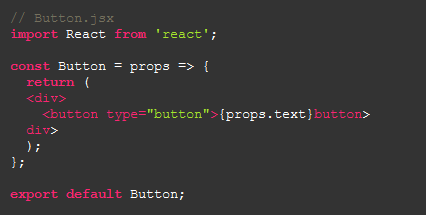
Estamos acostumbrados a escribir código HTML en archivos .html y la lógica de JavaScript en archivos .js.

React usa **JSX**: una sintaxis que nos permite escribir la estructura HTML y la lógica en JavaScript desde un mismo lugar: nuestros componentes.



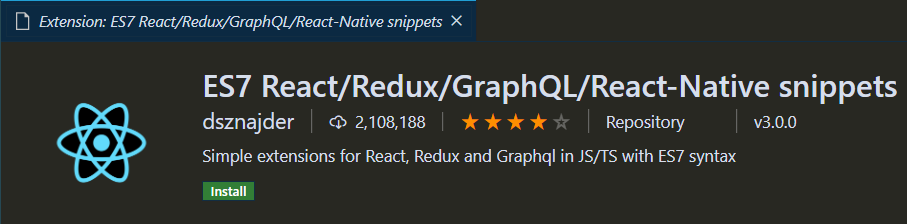
# Props: Comunicación entre Componentes

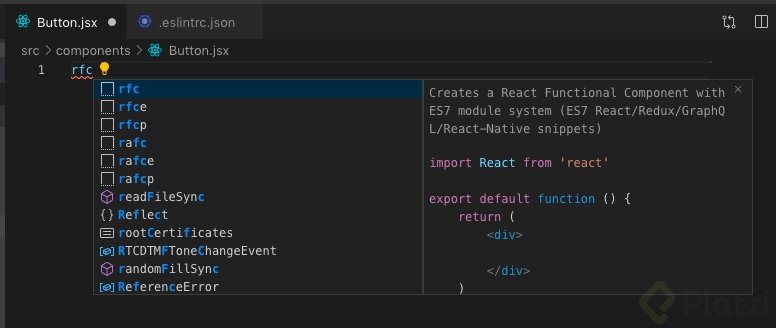
Las **Props** son la forma de enviar y recibir información en nuestros componentes. Son la forma de comunicar cada componente con el resto de la aplicación. Son muy parecidas a los parámetros y argumentos de las funciones en cualquier lenguaje de programación.



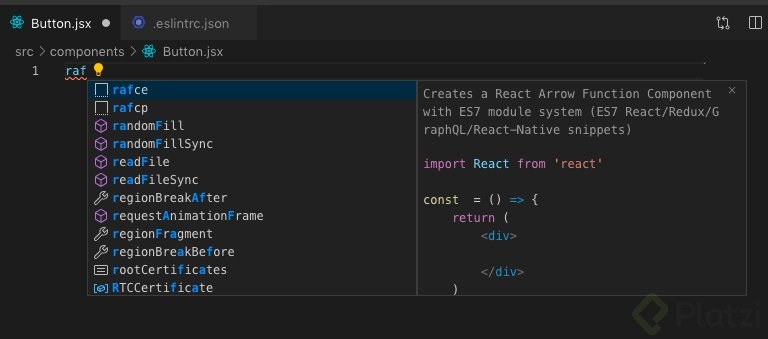


# Extension Útil





**React arrow functional component**



# ¿Qué son los métodos del ciclo vida?

## **¿Qué son los métodos del ciclo vida?**

Todos los componentes en React pasan por una serie de fases que generalmente se denominan “Ciclo de Vida del componente” es un proceso que React hace en cada componente, en algunos casos no podemos verlos como un bloque de código y en otros podemos llamarlos en nuestro componente para asignar una actividad según sea el caso necesario.

Los componentes en react pasan por un Montaje, Actualización, Desmontaje y Manejo de errores.

## **Montaje:**

En esta fase nuestro componente se crea junto a la lógica y los componentes internos y luego es insertado en el DOM.

## **Actualización:**

En esta fase nuestro componente está al pendiente de cambios que pueden venir a través de un cambio en “state” o “props” esto en consecuencia realizan una acción dentro de un componente.

## **Desmontaje:**

En esta etapa nuestro componente “Muere” cuando nosotros no necesitamos un elemento de nuestra aplicación, podemos pasar por este ciclo de vida y de esta forma eliminar el componente de la representación que tiene en el DOM.

## **Manejo de Errores:**

Cuando nuestro código se ejecuta y tiene un error, podemos entrar en una fase donde se puede entender mejor qué está sucediendo con la aplicación.

Algo que debemos tener en cuenta es que un componente NO debe pasar por toda las fases, un componente puede ser montado y desmontado sin pasar por la fase de actualización o manejo de errores.

Ahora que entendemos las fases que cumple el ciclo de vida en React vamos a entrar a detalle en cada uno de ellos para ver qué piezas de código se ejecutan y nos ayudarán a crear aplicaciones en React pasando por un ciclo de vida bien estructurado.

## Montado:

**Constructor()**

Este es el primer método al que se hace un llamado, aquí es donde se inicializan los métodos controladores, eventos del estado.

**getDerivedStateFromProps()**

Este método se llama antes de presentarse en el DOM y nos permite actualizar el estado interno en respuesta a un cambio en las propiedades, es considerado un método de cuidado, ya que su implementación puede causar errores sutiles.

**render()**

Si queremos representar elementos en el DOM en este método es donde se escribe esta lógica, usualmente utilizamos JSX para trabajar y presentar nuestra aplicación.

**ComponentDidMount()**

Este método se llama inmediatamente que ha sido montado en el DOM, aquí es donde trabajamos con eventos que permitan interactuar con nuestro componente.

## Actualización:

**getDerivedStateFromProps()**

Este método es el primero en ejecutarse en la fase de actualización y funciona de la misma forma que en el montaje.

**shouldComponentUpdate()**

Dentro de este método se puede controlar la fase de actualización, podemos devolver un valor entre verdadero o falso si queremos actualizar o no el componente y es utilizado principalmente para optimización.

**render()**

Se llama el método render que representa los cambios en el DOM.

**componentDidUpdate()**

Este método es invocado inmediatamente después de que el componente se actualiza y recibe como argumentos las propiedades y el estado y es donde podemos manejar nuestro componente.

## Desmontado

**componentWillUnmount()**

Este método se llama justo antes de que el componente sea destruido o eliminado del DOM.

## Manejo de Errores:

**getDerivedStateFromError()**

Una vez que se lanza un error este es el primer método que se llama, el cual recibe el error como argumento y cualquier valor devuelto en este método es utilizado para actualizar el estado del componente.

**componentDidCatch()**

Este método es llamado después de lanzarse un error y pasa como argumento el error y la información representada sobre el error.

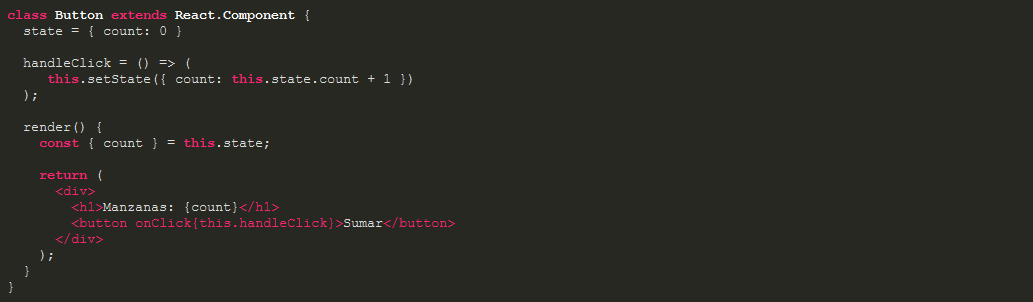
Ahora que entendemos cada una de las fases que tiene el ciclo de vida de react, podemos utilizarlas según sea necesario en nuestra aplicación y de esta forma crear las interacciones que necesitemos.

# 

# State - Events

React nos permite responder a las interacciones de los usuarios con propiedades como **onClick**, **onChange**, **onKeyPress**, **onFocus**, **onScroll**, entre otras.

Estas propiedades reciben el nombre de la función que ejecuta el código que responde a las interacciones de los usuarios. Seguramente, esta función usará la función **this.setState** para actualizar el estado de nuestro componente.



Recuerda que los nombres de estos eventos deben seguir la nomenclatura camelCase: primera palabra en minúsculas, iniciales de las siguientes palabras en mayúsculas y el resto también en minúsculas.

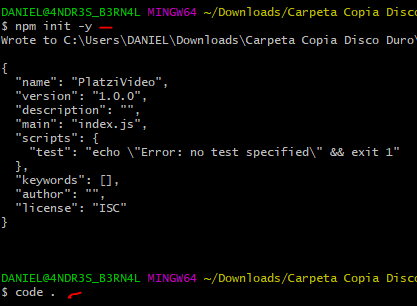
# Instalación y configuración de entorno.

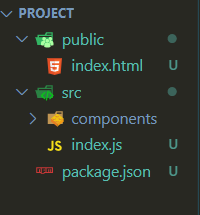
1. Iniciar un repositorio en GIT:



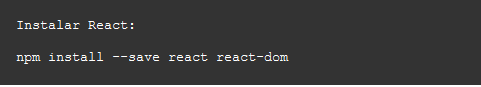
Iniciar un proyecto de Node.js:





Configuramos nuestras carpetas que queden de la siguiente manera.

Ahora vamos a instalar React en nuestro proyecto con el siguiente comando:



Usando React Inicialmente:

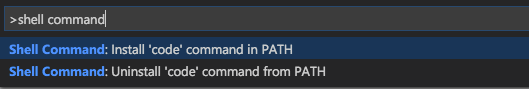




En caso de no funcionar:



En Visual Studio Code, abre tus configuraciones, selecciona "Shell Command: Install ‘code’ Command in PATH. Y listo! Deberías de poder abrir VSC desde tu terminal.



# Agregando compatibilidad con todos los navegadores usando Babel

**Babel** es una herramienta muy popular para escribir JavaScript moderno y transformarlo en código que pueda entender cualquier navegador.

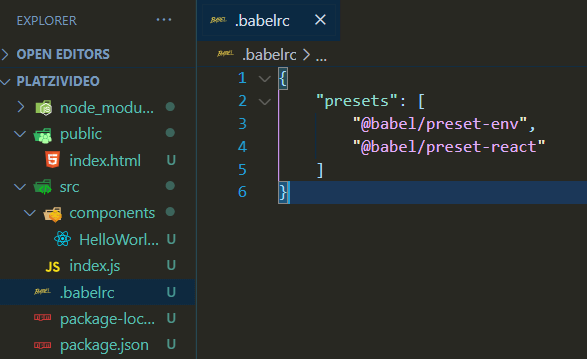
Instalación de Babel y otras herramientas para que funcione con React:

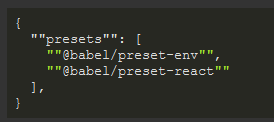


En caso previo de algún error durante el proceso:



Y volver a reintentar con el código inicial.

Luego crearemos un archivo en la raíz del proyecto con la Configuración de Babel (.babelrc):



**@babel/Preset-env :** Configura compatibilidad con ECMAScript 5 / 6

**@babel/Preset-react :** Configura compatibilidad con JSX y REACT

# Webpack: Empaquetando nuestros módulos.

**Webpack** es una herramienta que nos ayuda a compilar múltiples archivos (JavaScript, HTML, CSS, imágenes) en uno solo (o a veces un poco más) que tendrá todo nuestro código listo para producción.

Instalación de Webpack y algunos plugins:



Ahora creamos en nuestra raíz del proyecto (webpack.config.js) con la configuración de Webpack:

const path = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

  entry: './src/index.js',

  output: {

    path: path.resolve(\_\_dirname, 'dist'),

    filename: 'bundle.js',

  },

  resolve: {

    extensions: ['.js', '.jsx'],

  },

  module: {

    rules: [

      {

        test: /\.(js|jsx)$/,

        exclude: /node\_modules/,

        use: {

          loader: 'babel-loader',

        },

      },

      {

        test: /\.html$/,

        use: {

          loader: 'html-loader',

        },

      },

    ],

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: './public/index.html',

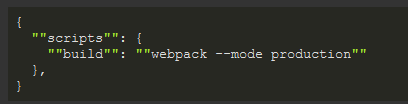
      filename: './index.html',

    }),

  ],

};

Luego creamos un Script en (package.json) para ejecutar las tareas de Webpack:

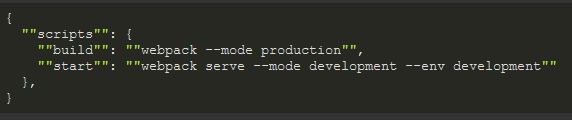


# Webpack Dev Server: Reporte de errores y Cambios en tiempo real

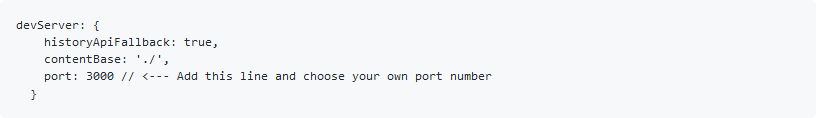
Instalación de **Webpack Dev Server**:



Script para ejecutar el servidor de Webpack y visualizar los cambios en tiempo real (package.json):



En caso de algún error relacionado al puerto agregar en el archivo webpack.config.js:



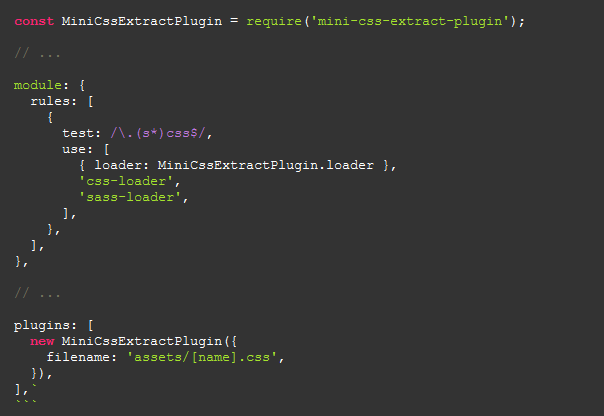
# Estilos con SASS

Los **preprocesadores** como **Sass** son herramientas que nos permiten escribir CSS con una sintaxis un poco diferente y más amigable que luego se transformará en CSS normal. Gracias a Sass podemos escribir CSS con variables, mixins, bucles, entre otras características.

Instalación de Sass:



Configuración de Sass en Webpack (webpack.config.js):



# Configuración final: ESLint y Git Ignore

El **Git Ignore** es un archivo que nos permite definir qué archivos NO queremos publicar en nuestros repositorios. Solo debemos crear el archivo **.gitignore** y escribir los nombres de los archivos y/o carpetas que no queremos publicar.

Para ello GitHub nos ofrece un ***gist*** que es un código de recomendación de lo que se debe dejar en este archivo.

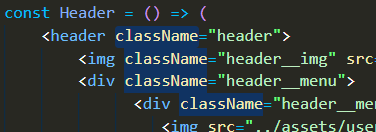
<https://gist.github.com/gndx/747a8913d12e96ff8374e2125efde544>

Los linters como **ESLint** son herramientas que nos ayudan a seguir buenas prácticas o guías de estilo de nuestro código.  
Se encargan de revisar el código que escribimos para indicarnos dónde tenemos errores o posibles errores. En algunos casos también pueden solucionar los errores automáticamente. De esta manera podemos solucionar los errores incluso antes de que sucedan.  
Instalación de ESLint:



Luego configuramos las reglas de ESLint en el archivo **.eslintrc** que debe ser creado en la raíz del proyecto. También tenemos un ***gist*** con el código que debemos añadir en él.

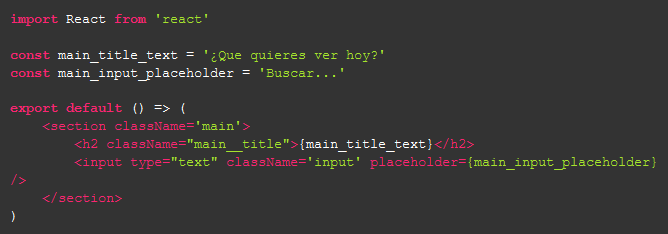
<https://gist.github.com/gndx/60ae8b1807263e3a55f790ed17c4c57a>

**Windows**  
Seleccionar todas las palabras iguales.   
**Ctrl+Shift+L**

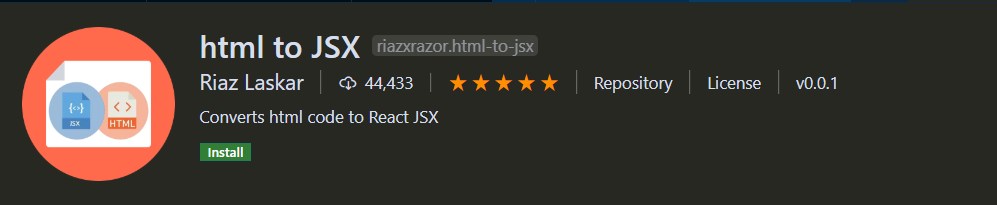
**Recomendación.**

Una buena práctica es no usar texto plano al HTML/JSX, es mejor pasarlo como variable para poder modificarlo con facilidad en el futuro.

**Ejemplo:**



**Extension Útil.**



# Añadiendo imágenes con Webpack

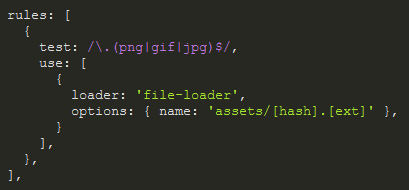
Vamos a usar **File Loader** para acceder a las imágenes de nuestro proyecto desde el código.

Inicialmente, estos archivos estáticos se encuentran junto al código de desarrollo. Pero al momento de compilar, Webpack guardará las imágenes en una nueva carpeta junto al código para producción y actualizará nuestros componentes (o donde sea que usemos las imágenes) con los nuevos nombres y rutas de los archivos.

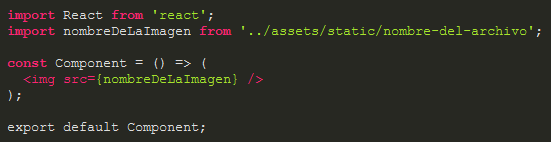
Instalación de File Loader:



Configuración de File Loader en Webpack (webpack.config.js):

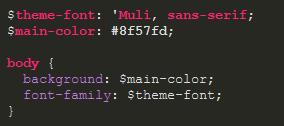


Uso de File Loader con React:

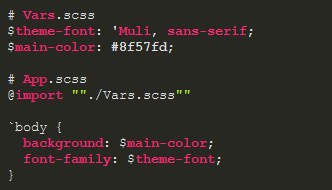


# Imports, Variables y Fuentes de Google en Sass

Así como JavaScript, Sass nos permite almacenar valores en variables que podemos usar en cualquier otra parte de nuestras hojas de estilo.



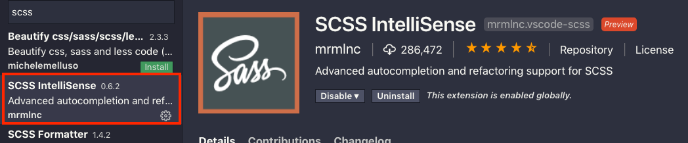
Podemos guardar nuestras variables en un archivo especial e importarlo desde los archivos de estilo donde queremos usar estas variables.



También podemos importar hojas de estilo externas a nuestra aplicación. Por ejemplo: las fuentes de Google.



En Visual Studio Code pueden descargar **SCSS IntelliSense** para el auto completado de las variables.  
**Extension:**



# Creando una Fake API

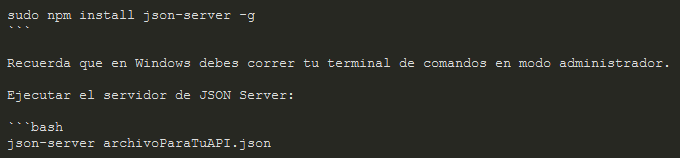
Vamos a usar **JSON Server** para crear una Fake API: una API ““falsa”” construida a partir de un archivo JSON que nos permite preparar nuestro código para consumir una API de verdad en el futuro.

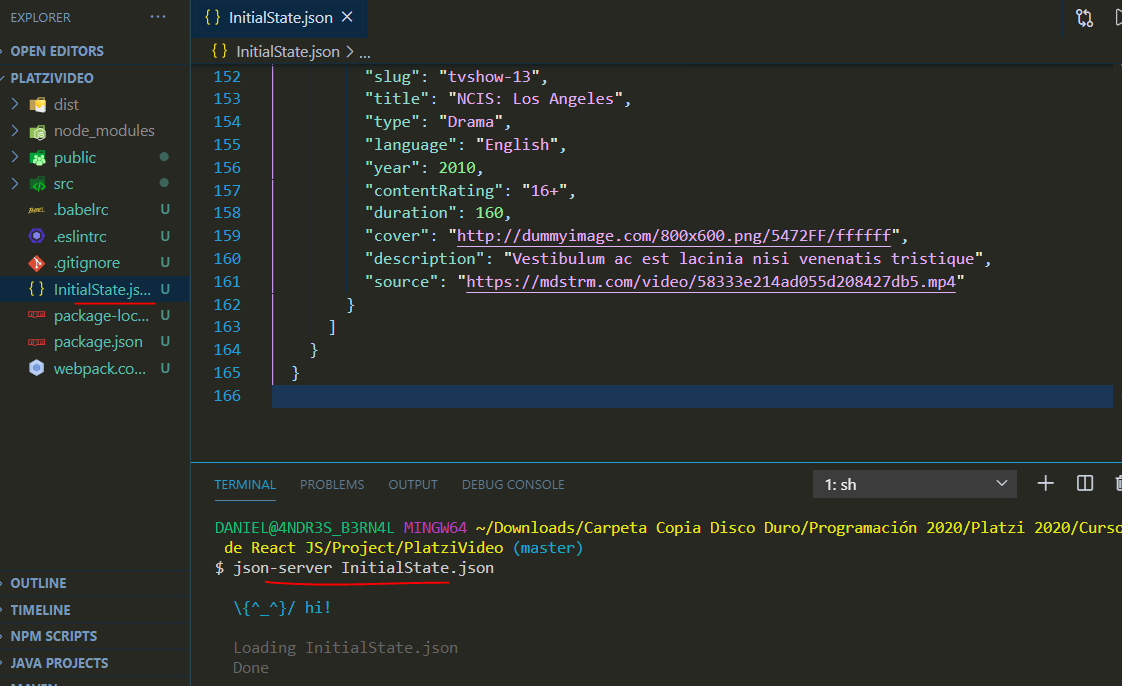
Este archivo lo llamamos **InitialState.js** en el que contendrá toda la información en JSON.

Ej:

<https://gist.github.com/gndx/d4ca4739450afaa614efe4570ac362ee>

Luego de la instalación del JSON Server iniciamos la API con json-server InitialState.js

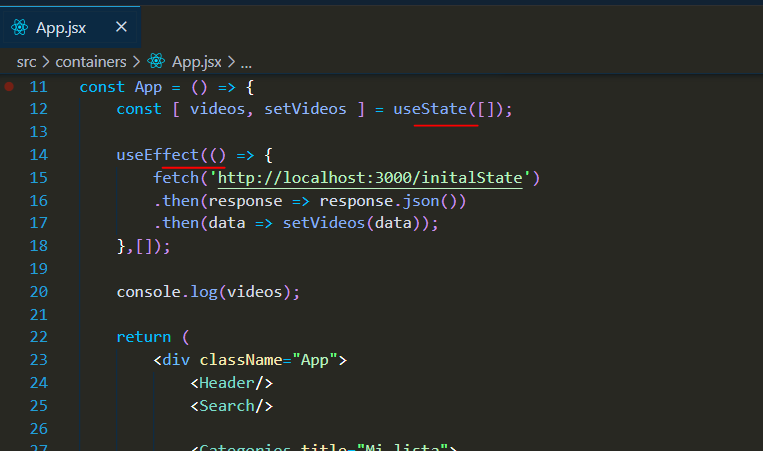




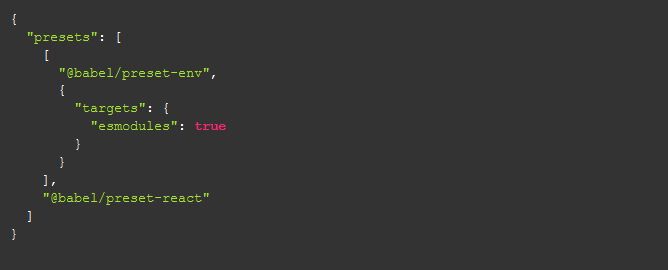
# React Hooks: useEffect y useState.

React Hooks es una característica de React disponible a partir de la versión 16.8 que nos permite agregar estado y ciclo de vida a nuestros componentes creados como funciones.

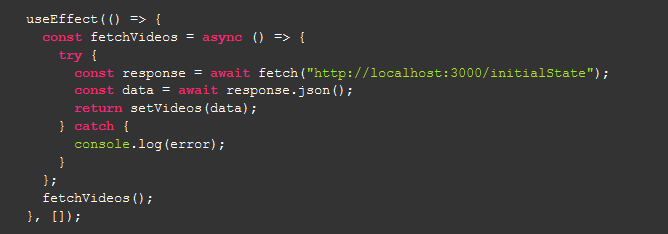
React es una librería desarrollada por Facebook que nos ayuda a construir interfaces de usuario interactivas para todo tipo de aplicaciones: páginas web, aplicaciones móviles o de escritorio, experiencias de realidad virtual, entre otras.



En el ejemplo anterior usamos Promesas. Pero si quisiéramos usar async/await en lugar de promesas, podemos hacerlo modificando el archivo .babelrc de esta manera:



Por otro lado, useEffect no puede recibir una función asíncrona (porque no puede devolver una promesa), pero podemos solucionarlo de así:



**En resumen:**

**useState** nos devuelve un array con dos elementos, el primero es el valor de nuestro estado, la segunda es una funcion que nos permite actualizar ese valor. El argumento que le enviamos es por defecto el valor del estado(initial state).

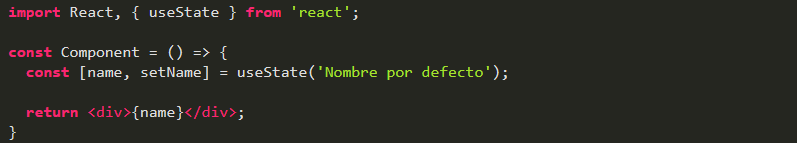
El hook **useEffect** es una funcion que nos permite ejecutar código cuando se monta, desmonta o actualiza un componente. El primer argumento que recibe es una funcion que se ejecutará cuando React monte o actualice el componente, esta funcion puede devolver otra funcion que se ejecuta cuando el componente se desmonte. El segundo argumento es una array donde podemos especificar que propiedades deben cambiar para que React vuelva a llamar nuestro código. Por defecto, cuando no enviamos un segundo argumento, React ejecutara useEffect cada vez que el componente o componente padre se actualice, sin enviamos un array vacío la funcion solo se ejecuta al montar o desmontar el componente.

**Lectura React Hooks**

Los **React Hooks** son una característica de React que tenemos disponible a partir de la versión 16.8. Nos permiten agregar estado y ciclo de vida a nuestros componentes creados como funciones.

El Hook **useState** nos devuelve un array con dos elementos: la primera posición es el valor de nuestro estado, la segunda es una función que nos permite actualizar ese valor.

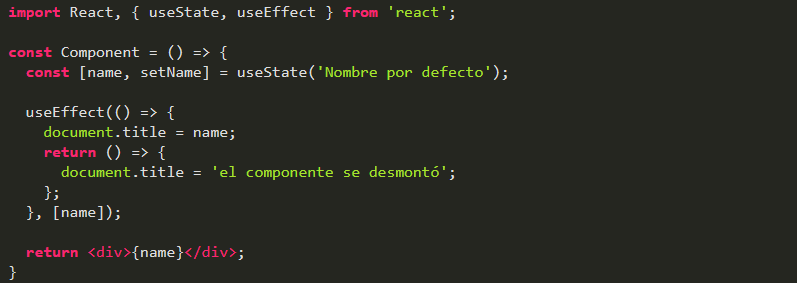
El argumento que enviamos a esta función es el valor por defecto de nuestro estado (initial state).

 El Hook **useEffect** nos permite ejecutar código cuando se monta, desmonta o actualiza nuestro componente.

El primer argumento que le enviamos a useEffect es una función que se ejecutará cuando React monte o actualice el componente. Esta función puede devolver otra función que se ejecutará cuando el componente se desmonte.

El segundo argumento es un array donde podemos especificar qué propiedades deben cambiar para que React vuelva a llamar nuestro código. Si el componente actualiza pero estas props no cambian, la función no se ejecutará.

Por defecto, cuando no enviamos un segundo argumento, React ejecutará la función de useEffect cada vez que el componente o sus componentes padres actualicen. En cambio, si enviamos un array vacío, esta función solo se ejecutará al montar o desmontar el componente.

 No olvides importar las funciones de los hooks desde la librería de React. También puedes usarlos de esta forma: **React.useNombreDelHook**.

Lectura para reforzar:

<https://desarrollofront.medium.com/entendiendo-los-hooks-de-react-c%C3%B3mo-usar-usestate-y-useeffect-en-nuestros-componentes-611b9e826dfa>

# Custom Hooks

React nos permite crear nuestros propios Hooks. Solo debemos seguir algunas convenciones:

* Los hooks siempre deben empezar con la palabra use: useAPI, useMovies, useWhatever.
* Si nuestro custom hook nos permite consumir/interactuar con dos elementos (por ejemplo, title y setTitle), nuestro hook debe devolver un array.
* Si nuestro custom hook nos permite consumir/interactuar con tres o más elementos (por ejemplo, name, setName, lastName, setLastName, etc.), nuestro hook debe devolver un objeto.

<https://es.reactjs.org/docs/hooks-custom.html>

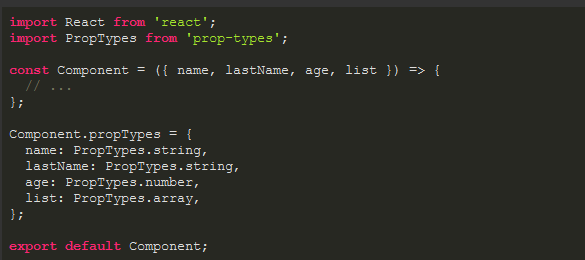
# PropTypes

Los **PropTypes** son una propiedad de nuestros componentes que nos permiten especificar qué tipo de elementos son nuestras props: arrays, strings, números, etc.

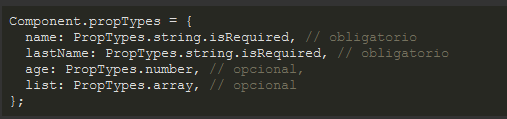
Instalación de PropTypes:



Uso de PropTypes:

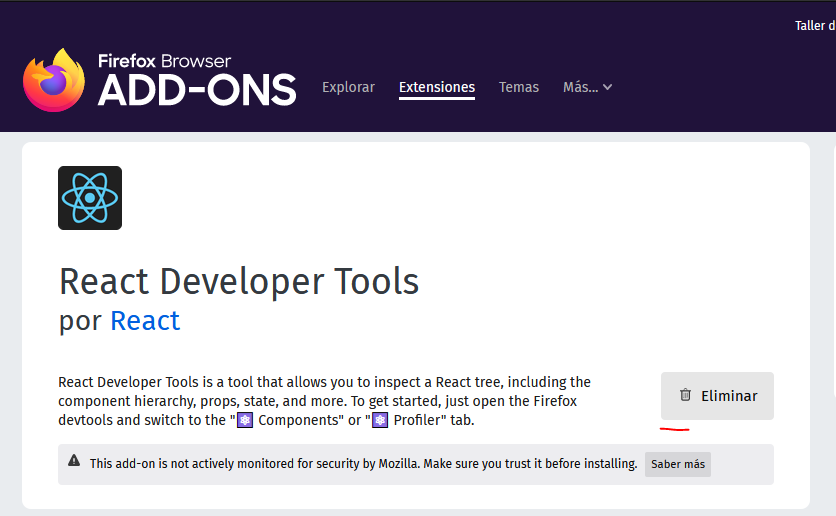


Por defecto, enviar todas nuestras props es opcional, pero con los propTypes podemos especificar cuáles props son obligatorias para que nuestro componente funcione correctamente con el atributo isRequired.



# Debuggeando React con React DevTools

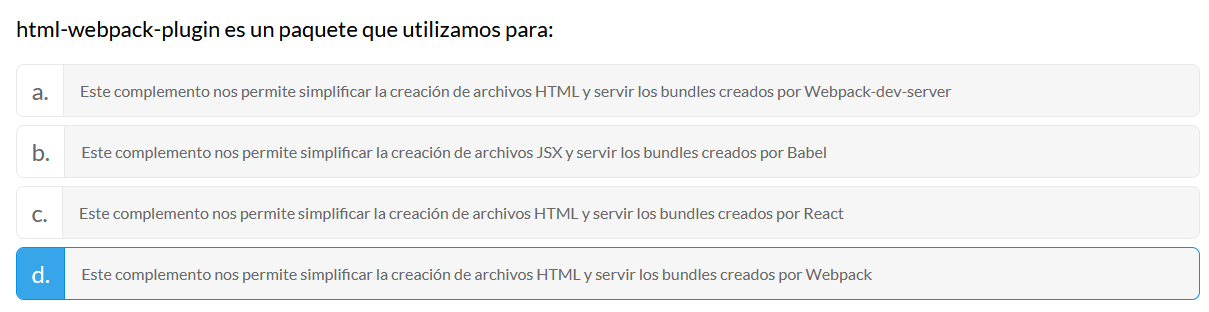
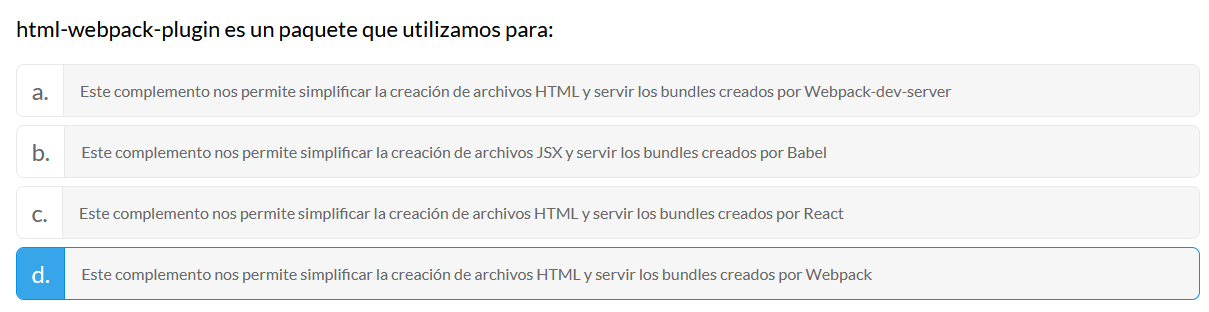
**React DevTools** es una herramienta muy parecida al Inspector de Elementos. Nos permite visualizar, analizar e interactuar con nuestros componentes de React desde el navegador.

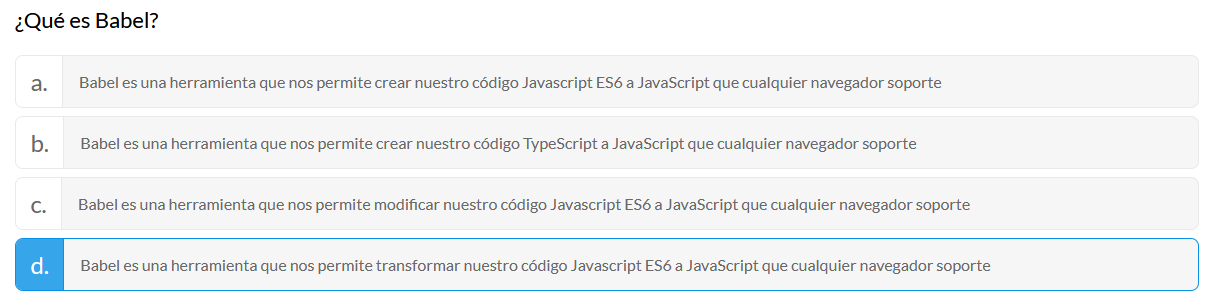
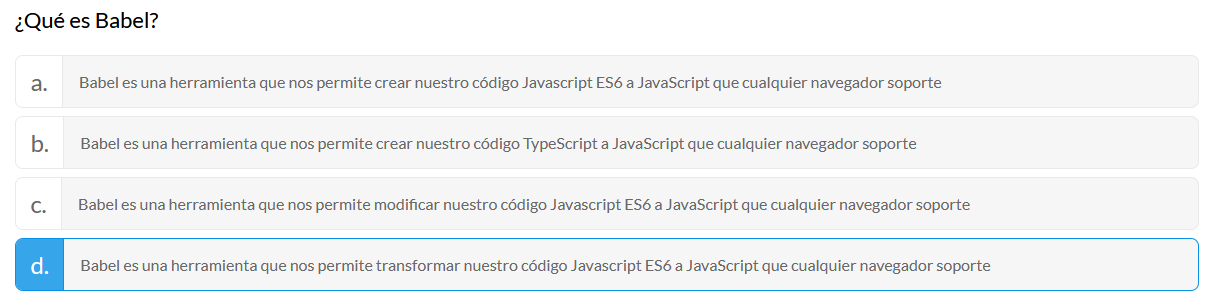


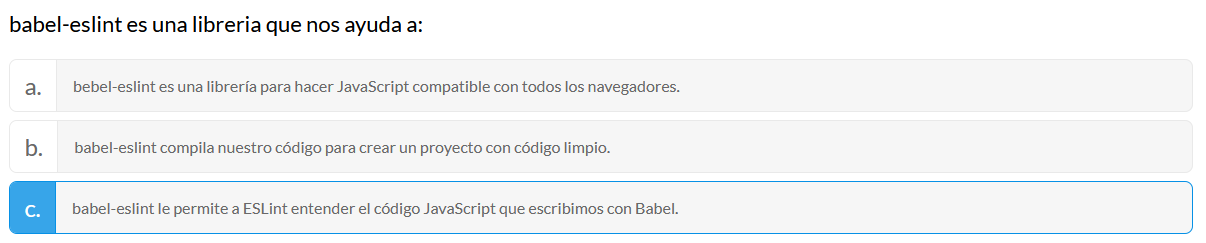
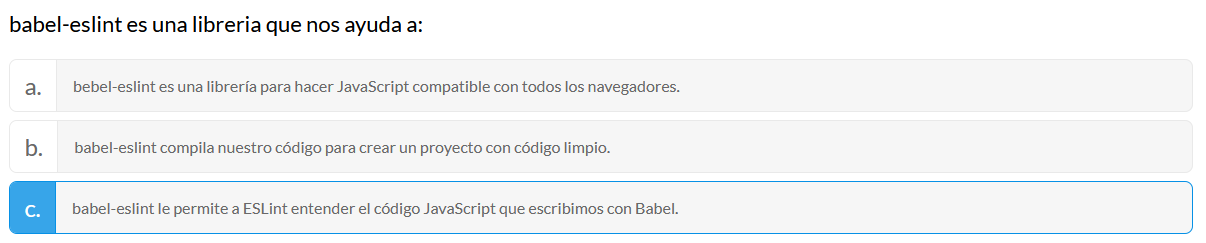
<https://addons.mozilla.org/es/firefox/addon/react-devtools/>

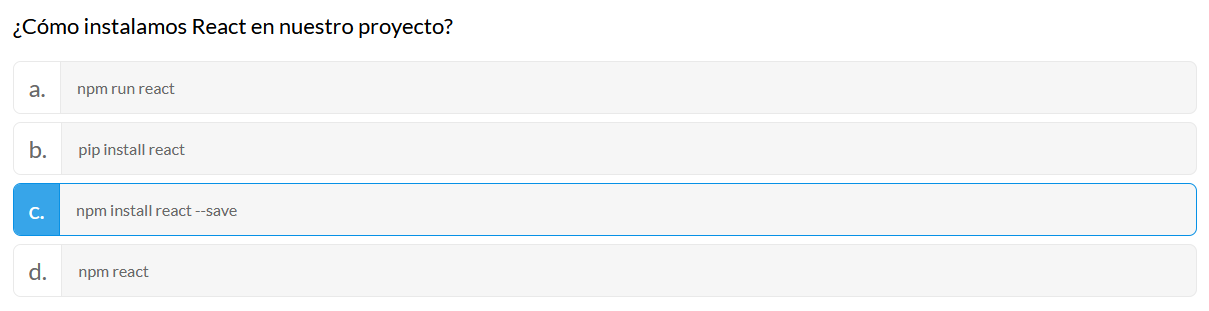
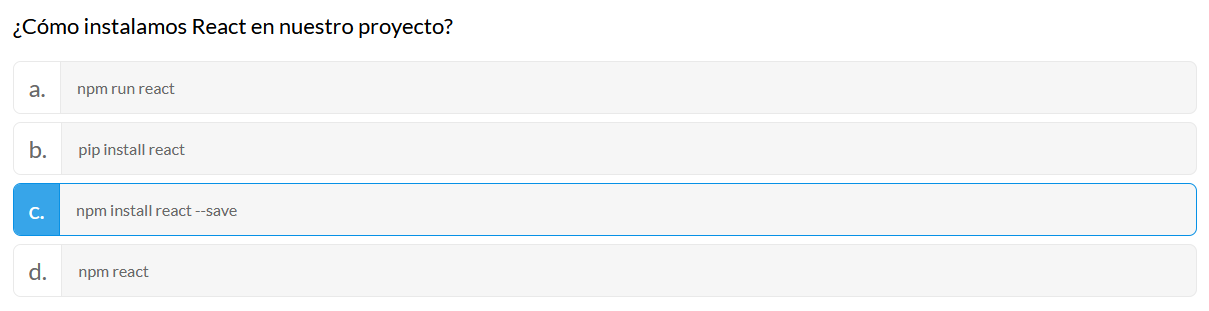
Importants Questions and Answers.

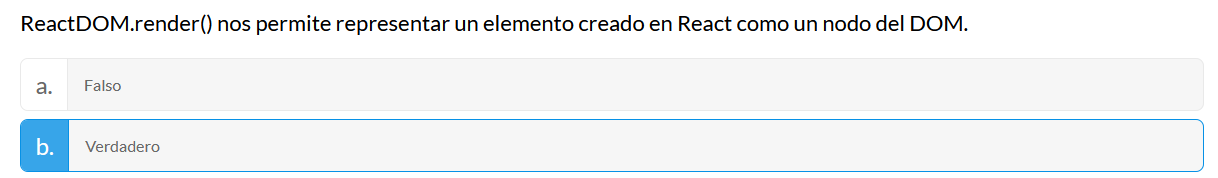
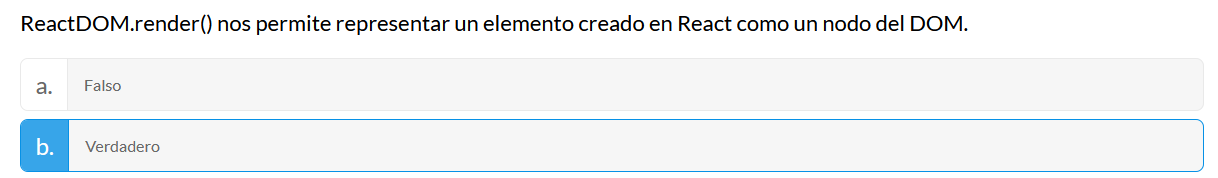


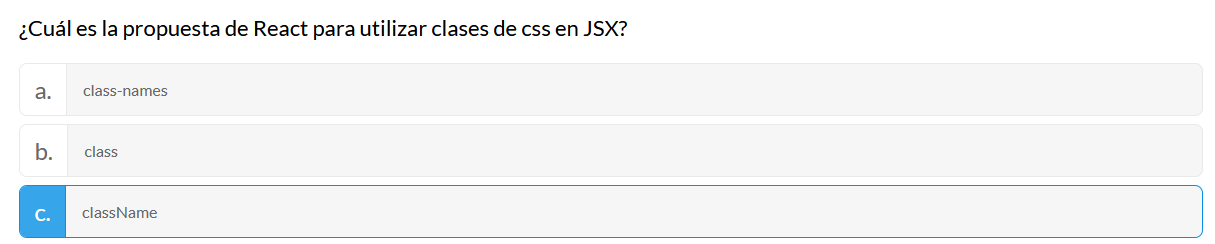
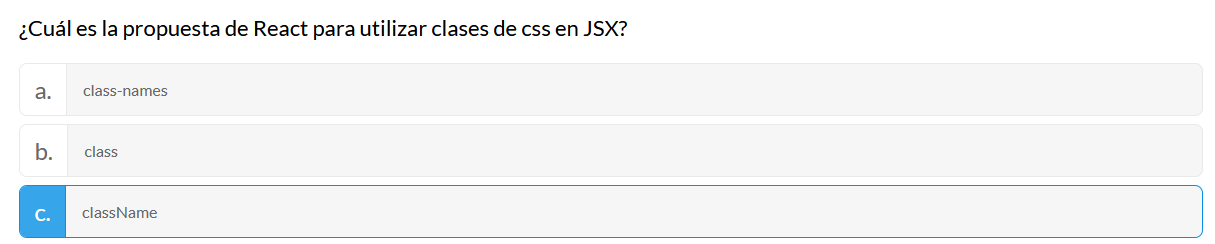


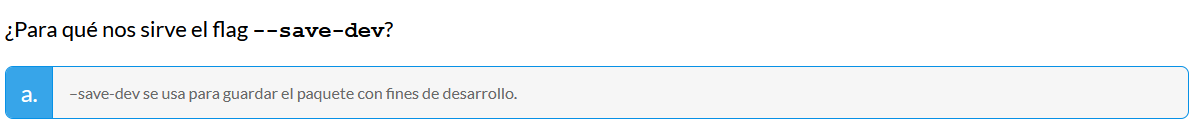


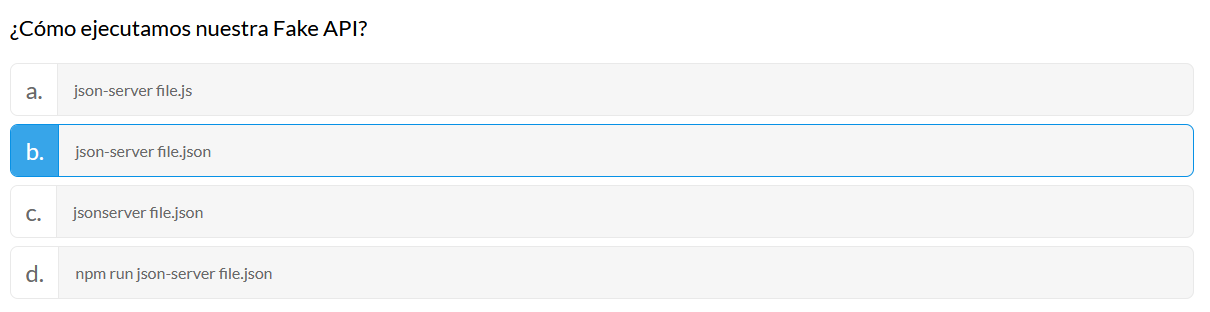
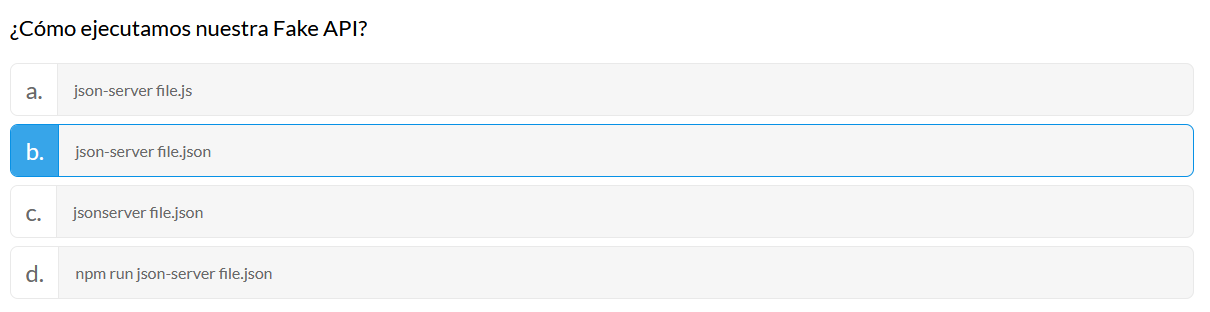


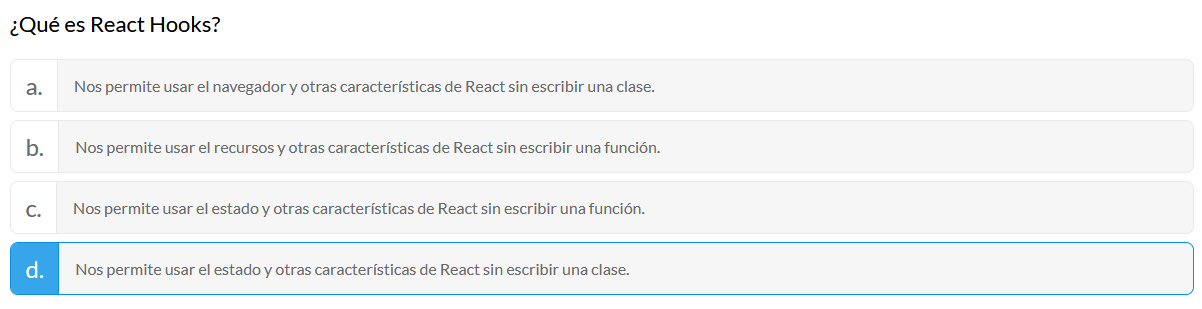
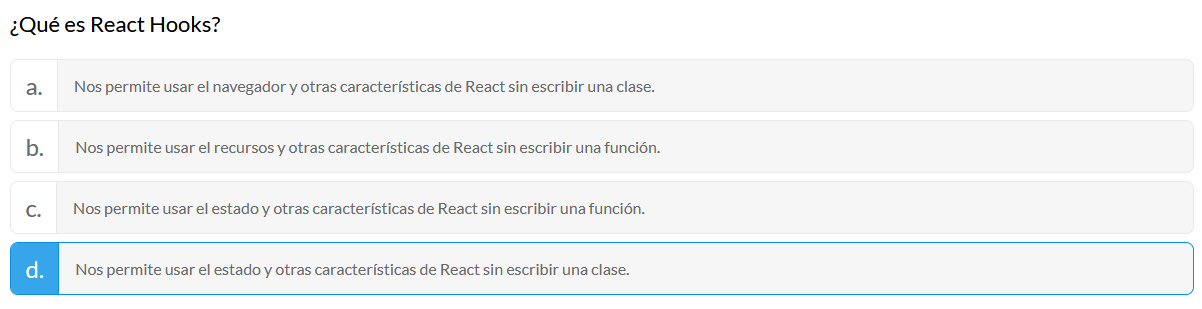


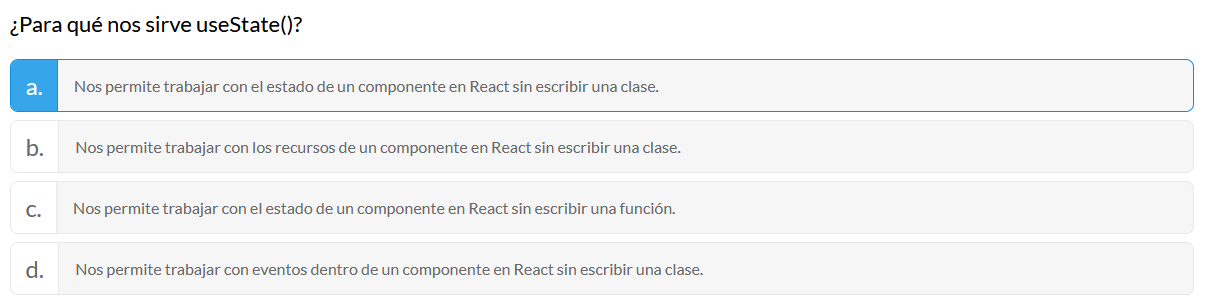


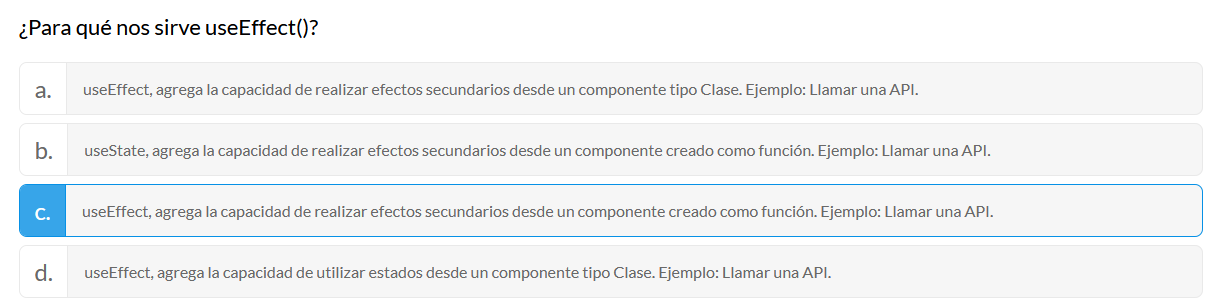
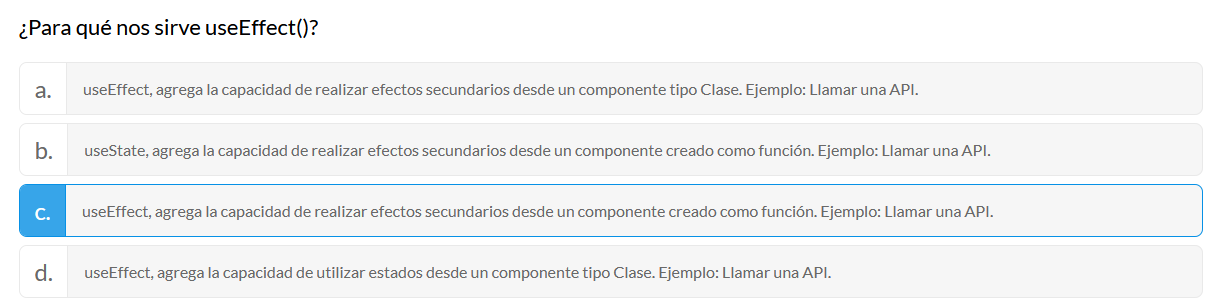


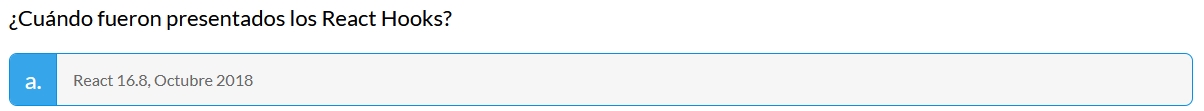


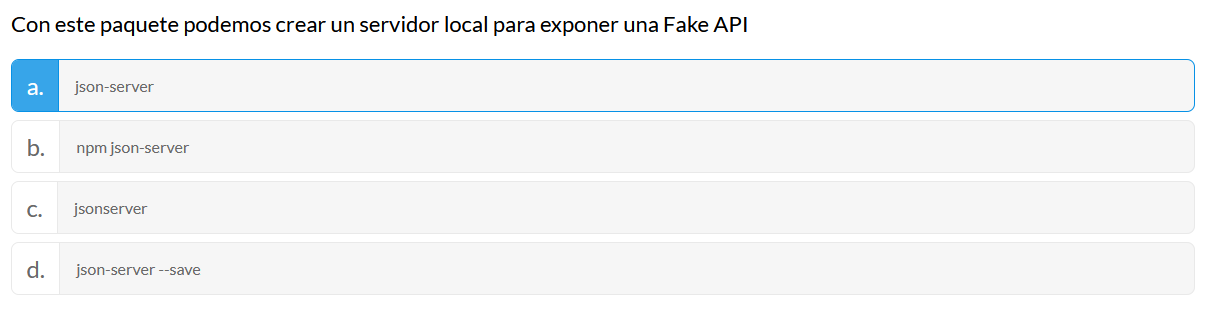


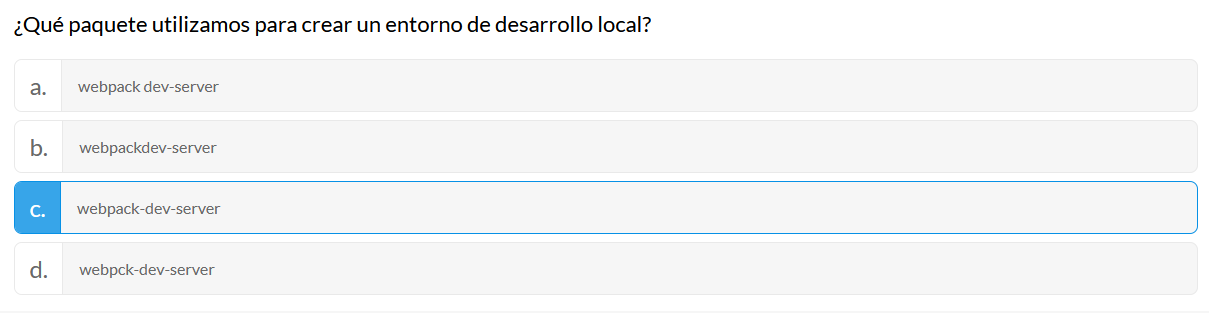
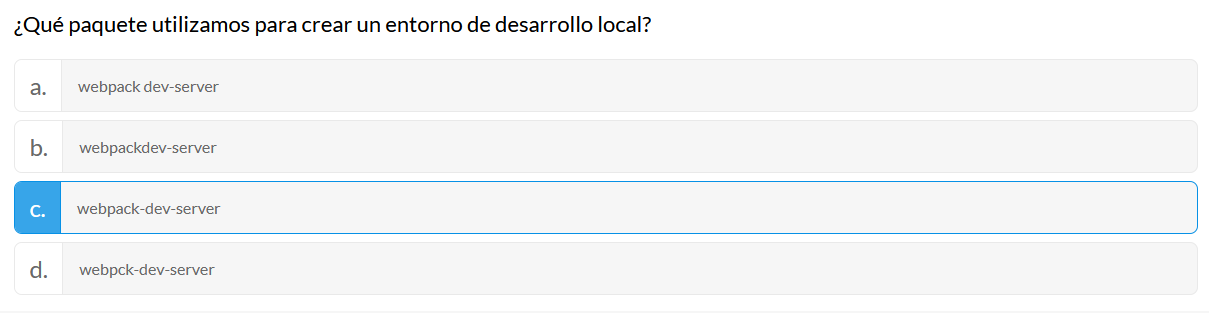


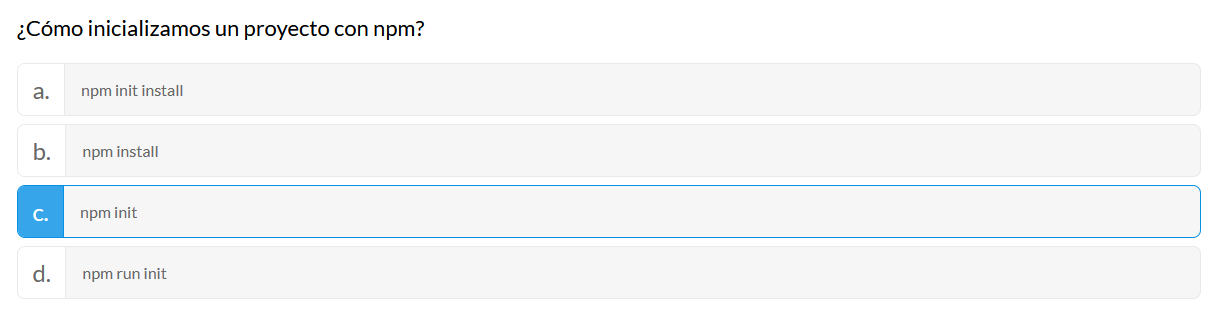
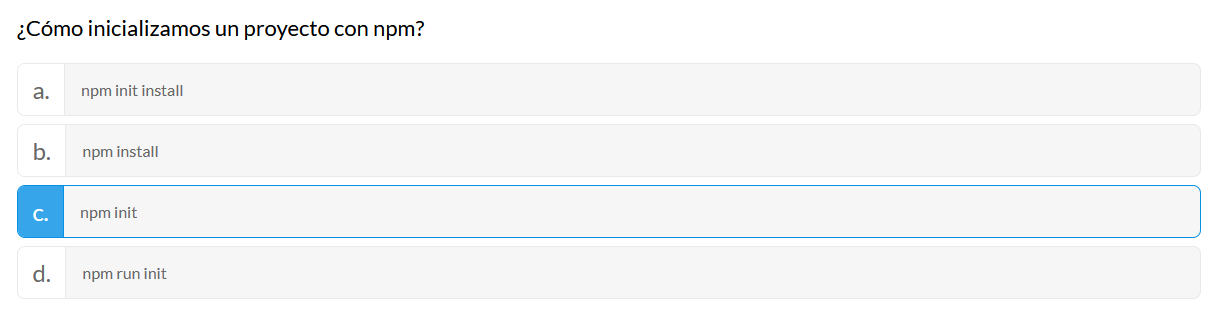


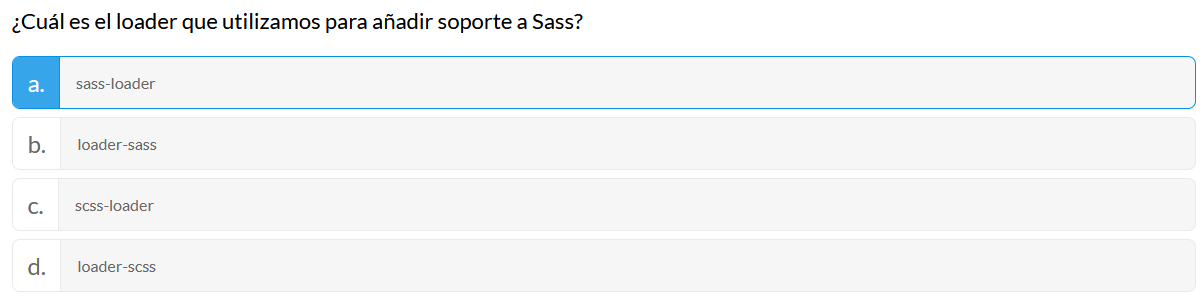


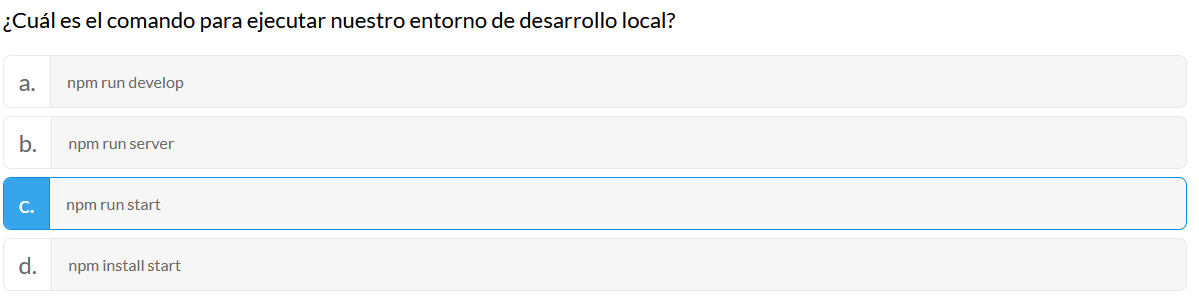
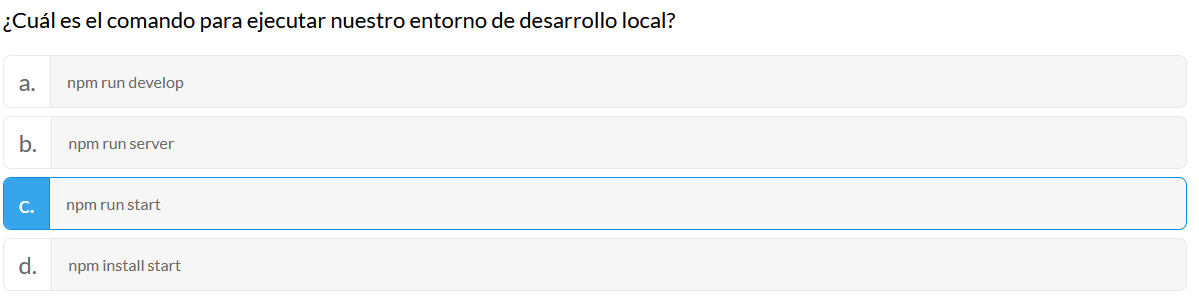


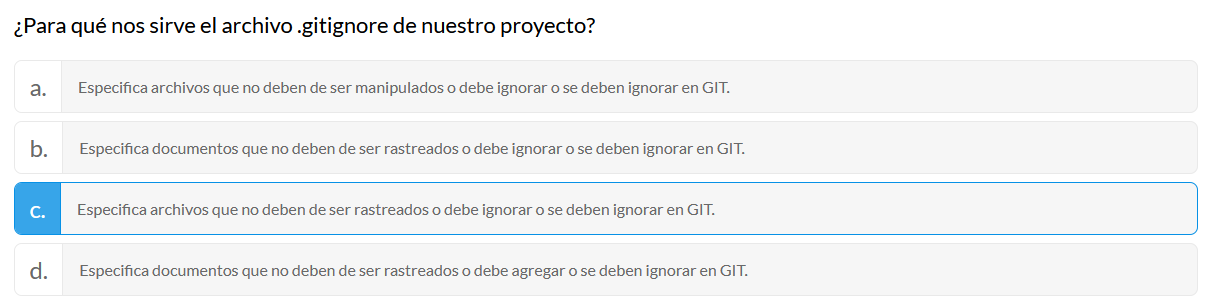
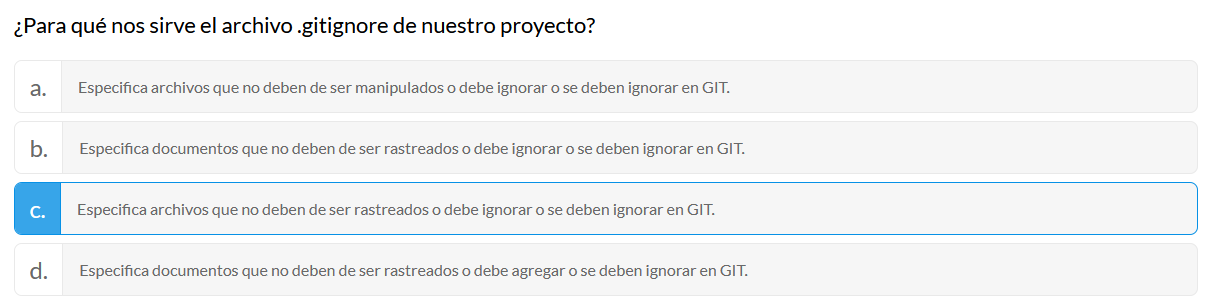


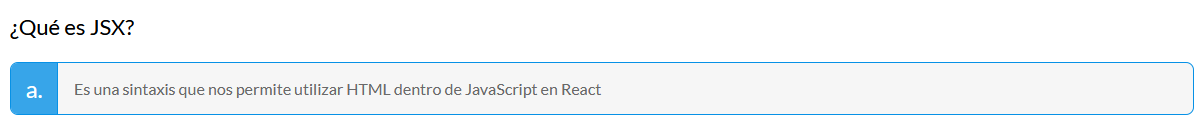




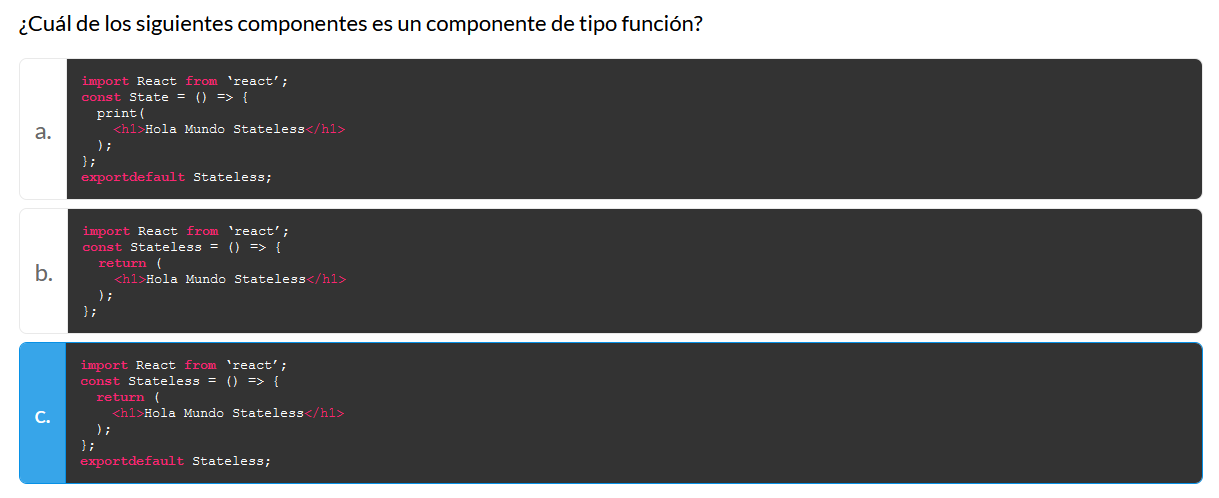
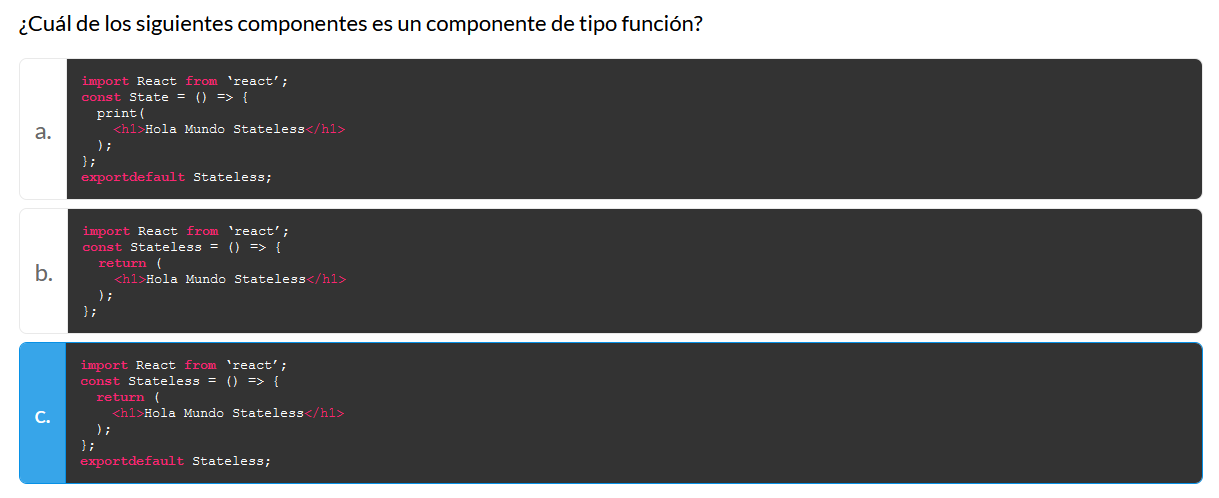


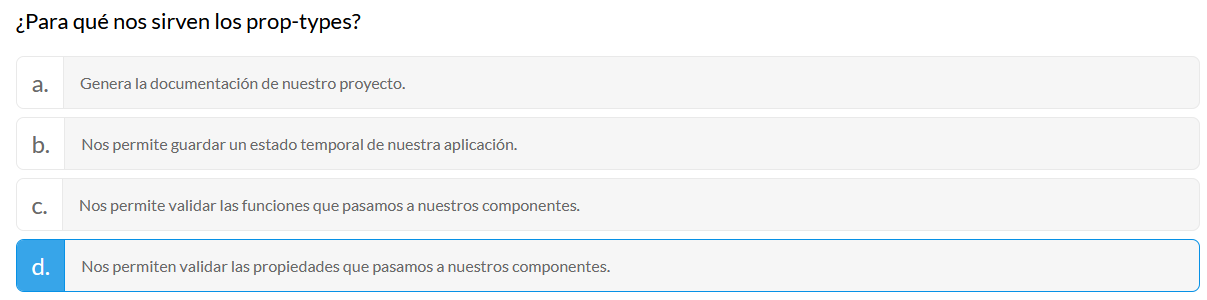
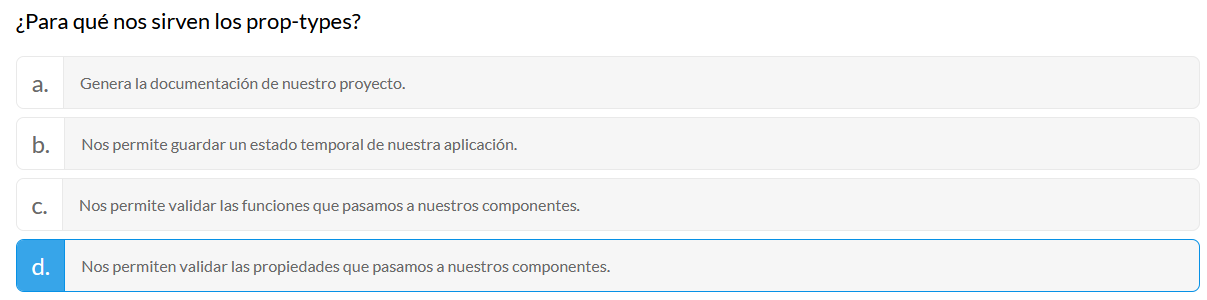












**Curso de React JS.**

**¿Qué es React.js?**

**React** cumple su función como *biblioteca* ya que para utilizar su código se debe importar. También es un *Framework* aunque las convenciones de cómo debe ser organizado todo no son estrictas.  
En este curso aprenderás las prácticas que la comunidad ha decidido que son buenas.

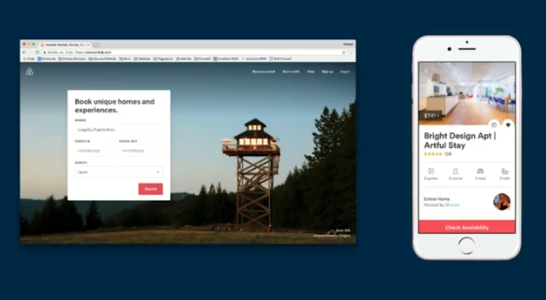
**React es declarativo**, lo que quiere decir que se le indica qué debe hacer pero no cómo debe hacerse, ahorrando de esta manera muchos pasos.



**JSX** es HTML dentro de Javascript, esto se verá más adelante en detalle.

React está estructurado por **componentes** que son como pequeños bloques de lego que al ser unidos forman aplicaciones de React. Estos componentes pueden tener estilos, ser enlazados a eventos y sus estados pueden ser modificados.

Con React también se tiene la ventaja de que será escrito una sola vez y podrá ser utilizado en aplicaciones web, móviles, entre otras.



# Herramientas que usaremos.

Estas son las herramientas que usaremos en el curso:

* **Navegador:** Especialmente Chrome ya que cuenta con óptimas herramientas de desarrollo.
* **React Developer Tools:** Es una herramienta Open Source creada por Facebook y tiene instalación para Chrome o Firefox. Nos dejará ver el código de React inspeccionando elementos.
* **Editor de texto:** Puedes usar cualquiera, en este curso sugerimos **Visual Studio Code**. Tiene muchos plugins útiles para el desarrollo.
* **Prettier:** Es un plugin que hace que el código se vea bien sin importar cómo está escrito.

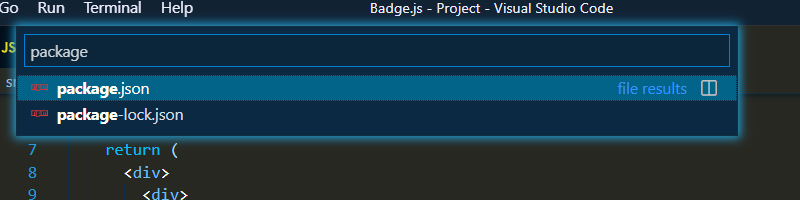
# 

# Importante tener configurado la opción siguiente para que así haga el autoformateado cuando guardan el archivo.

# 

Agregar Autocompletado de HTML en JSX.

Para los que usan Visual Studio Code y desean autocompletar código JSX vayan a **File - Preferences - Settings** y busquen el archivo settings.json.



Para abrir nuestra ventana para buscar el archivo Package.json oprimimos:

(Ctrl+Shift+O)

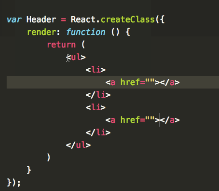
Luego Agregamos el siguiente código dentro del archivo:

    "emmet.includeLanguages": {

        "javascript": "javascriptreact"

      },

      "emmet.triggerExpansionOnTab": true



**Create-react-app**

**Create-react-app** es una aplicación moderna que se usa desde una línea de comando. Antes de ella se configuraba todo el entorno manualmente lo cual tomaba mucho tiempo.

**Pasos para obtenerlo:**  
Se debe instalar desde la línea de comando usando  
npm install -g create-react-app

Una vez instalado se crea la carpeta del proyecto con  
create-react-app -nombre del proyecto-

En este punto se estará instalando React y otras herramientas, también se configurará el entorno usando **Webpack**.

Una vez se instala todo entra a la carpeta *src* donde estará todo el código fuente de la aplicación, siendo el más importante *index.js* que es el punto de entrada a la aplicación.

Finalmente para correr la aplicación se usa el comando  
npm run start

**Otras herramientas:**

* **Babel:** Traduce Javascript moderno (JSX) a un Javascript que todos los navegadores puedan interpretar.
* **Eslint:** Lee el código y avisa de errores.

# Ejemplo: Clonar un código de GitHub

En esta clase vamos a comenzar clonando el código del proyecto del repositorio en GitHub.

Es importante que tú y yo tengamos un punto de partida en común. Así vamos a poder asegurarnos que cada cambio que yo haga en el código, tú también lo recibas.

Para hacer esto, en la terminal ve a una carpeta donde quieras que exista el proyecto. Entonces escribes lo siguiente:

$ git clone https://github.com/Sparragus/platzi-badges.git

Eso va a clonar el repositorio del curso a una carpeta llamada platzi-badges.

Ahora es necesario que te muevas a esa carpeta.

$ cd platzi-badges

Ahora necesitamos instalar todas las dependencias necesarias para poder correr el proyecto. Lo haremos utilizando npm.

$ npm install

Este proceso puede tardar un poco. Lo que está haciendo es descargando todas las bibliotecas de código que el proyecto necesita.

Una vez haya concluido, estamos listos para echar a correr el servidor. Lo hacemos con el comando

$ npm run start

Cuando el servidor comience, automáticamente va a abrir una pantalla en el navegador con la aplicación.

Si todo salió bien, vas a ver una pantalla que dice “Hello, Platzi Badges”.

En la próxima clase vas a aprender como fue que “Hello, Platzi Badges” llegó desde el código hasta la pantalla de tu navegador.

\*[Aquí](https://github.com/sparragus/platzi-badges) encuentras el repositorio.

**ReactDOM.render**

* **React** y **ReactDOM** trabajarán en conjunto.
  + React como análogo a **createElement**
  + ReactDOM a **appendChild**
* **ReactDOM.render()** toma dos argumentos: Qué queremos renderizar y dónde lo queremos renderizar.
* Siempre que escribas **JSX** es requisito importar **React.**



React es JavaScript, pero con React vamos a tener ventajas a la hora escribir nuestro código como por ejemplo: la manera en que Javascript normalmente crea un elemento es con (**createElement(‘h1’)**) y para añadir ese elemento a un contenedor usamos (**appendChild()**). Pero con React vamos a tener la ventaja de usar **ReactDOM.render(),** para realizar de manera más fácil esto mismo.

Al importar react, vamos a poder hacer uso de **JSX** que nos permite usar HTML dentro de JS sin problemas.

import React from 'react';

import ReactDOM from 'react-dom';

**JSX**

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

JSX tiene su alternativa que es **React.createElement** pero es preferible JSX porque es mucho más legible y expresivo. Ambos tienen el mismo poder y la misma capacidad.

**React.createElement** recibe 3 argumentos:

* El tipo de elemento que estamos creando
* sus atributos o *props*
* y el *children* que es el contenido.

Ejemplo:  
React.createElement(‘a’, { href: ‘https://platzi.com’ }, ‘Ir a Platzi’);

En JSX se utilizan las llaves para introducir variables o expresiones de Javascript. Lo que sea que esté adentro se va a evaluar y su resultado se mostrará en pantalla.

Las expresiones pueden ser llamadas a otras funciones, cálculos matemáticos, etc. Si las expresiones son false, 0, null, undefined, entre otros, no se verán.



**¿Qué es un componente?**

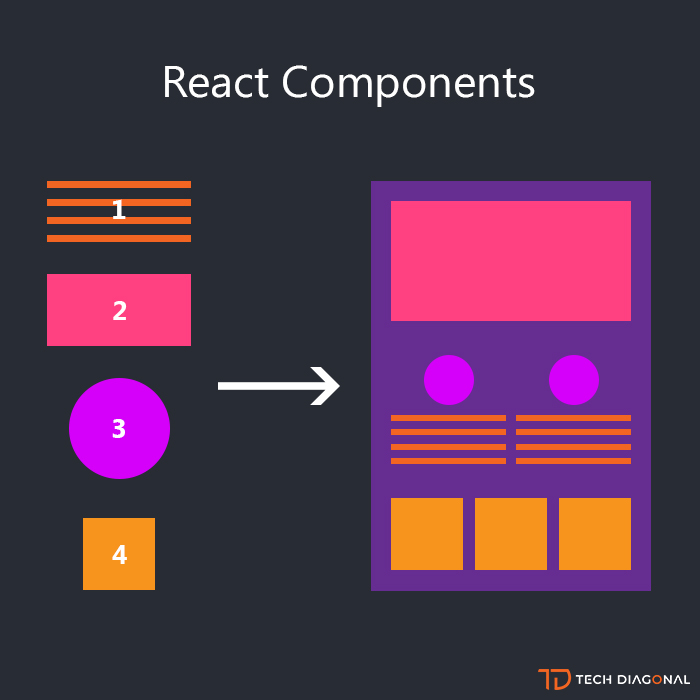
Los componentes en React son **bloques de construcción**.  
Las aplicaciones hechas con React son como figuras de Lego. Junta varias piezas (componentes) y puedes construir un website tan pequeño o tan grande como quieras.  
Los componentes serán barras de búsquedas, enlaces, encabezados, el *header*, etc.

**”Componente” vs “elemento**  
Un elemento es a un objeto como un componente es a una clase. Si el elemento fuera una casa, el componente serían los planos para hacer esa casa.

**Identificación de componentes**  
Para identificarlos debes hacerte las siguientes preguntas:

* ¿Qué elementos se repiten? Estos son los elementos en una lista o los que comparten aspecto visual y su funcionalidad
* ¿Qué elementos cumplen una función muy específica? Estos sirven para encapsular la lógica y permiten juntar muchos comportamientos y aspectos visuales en un solo lugar.

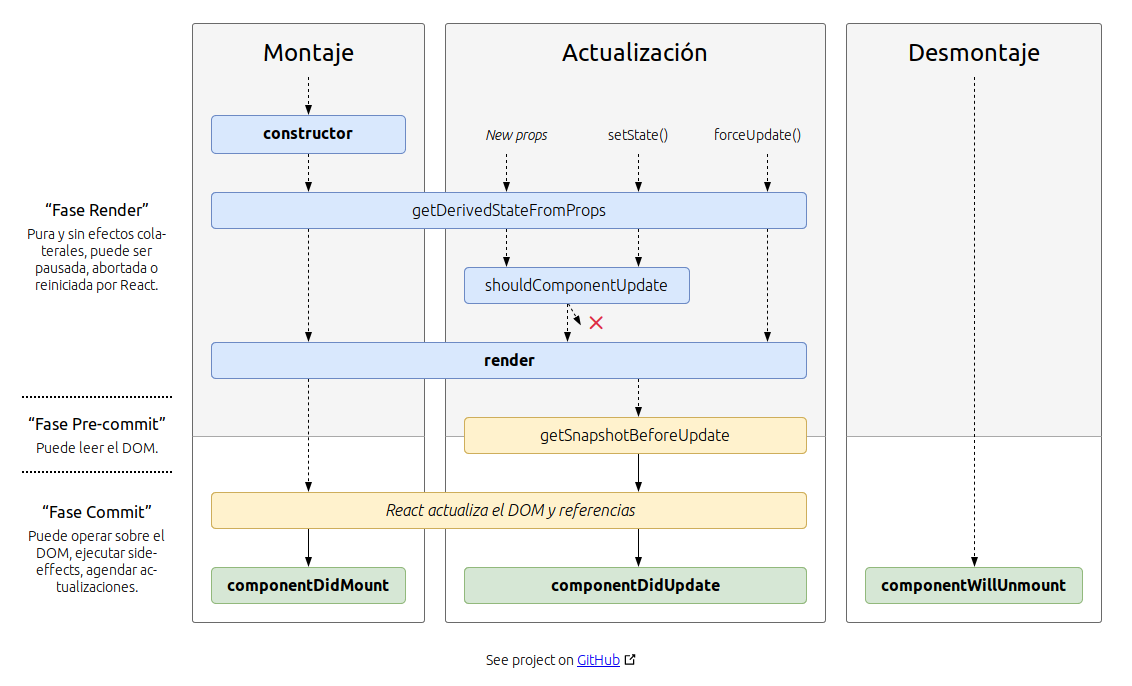
**Identificar componentes es una habilidad esencial para poder desarrollar aplicaciones de React.**



# Qué es y cómo funciona un componente en React.js

En esta clase aprenderás acerca del ciclo de vida de los componentes en React para crear aplicaciones dinámicas. Desde la importancia del montaje cuando los usuarios llegan por primera vez a nuestra aplicación, hasta la actualización y desaparición de los componentes.

Los **componentes en React tienen vida; nacen, crecen y desaparecen** .

El **ciclo de vida de los componentes tiene 3 fases** :  
1- El **Montaje** es cuando los usuarios llegan a nuestra aplicación, cuando tienen su 1era interacción con él.  
2- La **Actualización** , es cuando **se ejecuta el render**, **generando el nuevo DOM**, es cuando React manda una **señal de actualización componentDidUpdate()** .  
3- **Eliminación** de los componentes, al entrar a otra página, varios componentes no estarán en ella, React manda la señal **componentWIllUnmount()**, seguido de la **eliminación del código en el DOM**.

<https://medium.com/@jmz12/entendiendo-los-ciclos-de-vida-8a70abb3b51a>

**Creación de un componente**

* Es una buena práctica que los componentes vivan en su propio archivo y para ello se les crea una carpeta.
* Todos los componentes requieren por lo menos el método **render** que define cuál será el resultado que aparecerá en pantalla.
* El *source* de las imágenes en React puede contener direcciones en la web o se le puede hacer una referencia directa importándola. Si se importa deben usarse llaves para que sea evaluado.



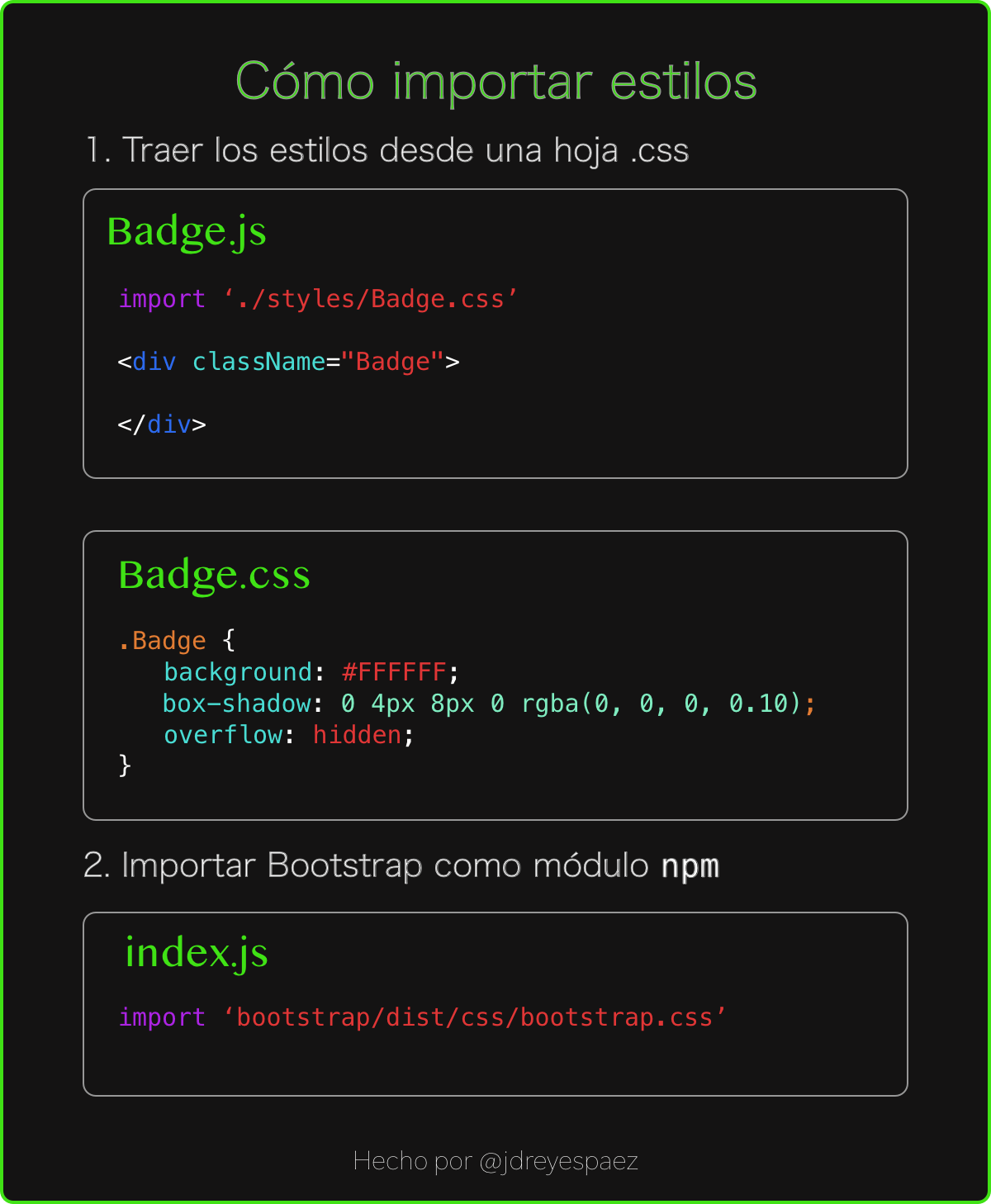
**API Útil para imágenes de Avatar.**

<https://www.gravatar.com/avatar?d=identicon>



**Cómo aplicar estilos**

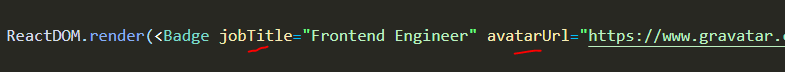
* Para los estilos crearemos una carpeta llamada **Styles** y allí vivirán todos los archivos de estilos que tienen que ver con los componentes.
* Para usar los estilos es necesario importarlos con *import*
* React funciona ligeramente diferente y para los atributos de clases no se utiliza *class* sino *className*
* Es posible utilizar **Bootstrap** con React, sólo debe ser instalado con npm install bootstrap y debe ser importado en el **index.js**
* Existen estilos que son usados de manera global o en varios componentes, así que deben ser importados en el index.js



# Props

Los **props** que es la forma corta de properties son argumentos de una función y en este caso serán los atributos de nuestro componente como **class**, **src**, etc.

Estos props salen de una variable de la clase que se llama this.props y los valores son asignados directamente en el **ReactDOM.render()**.





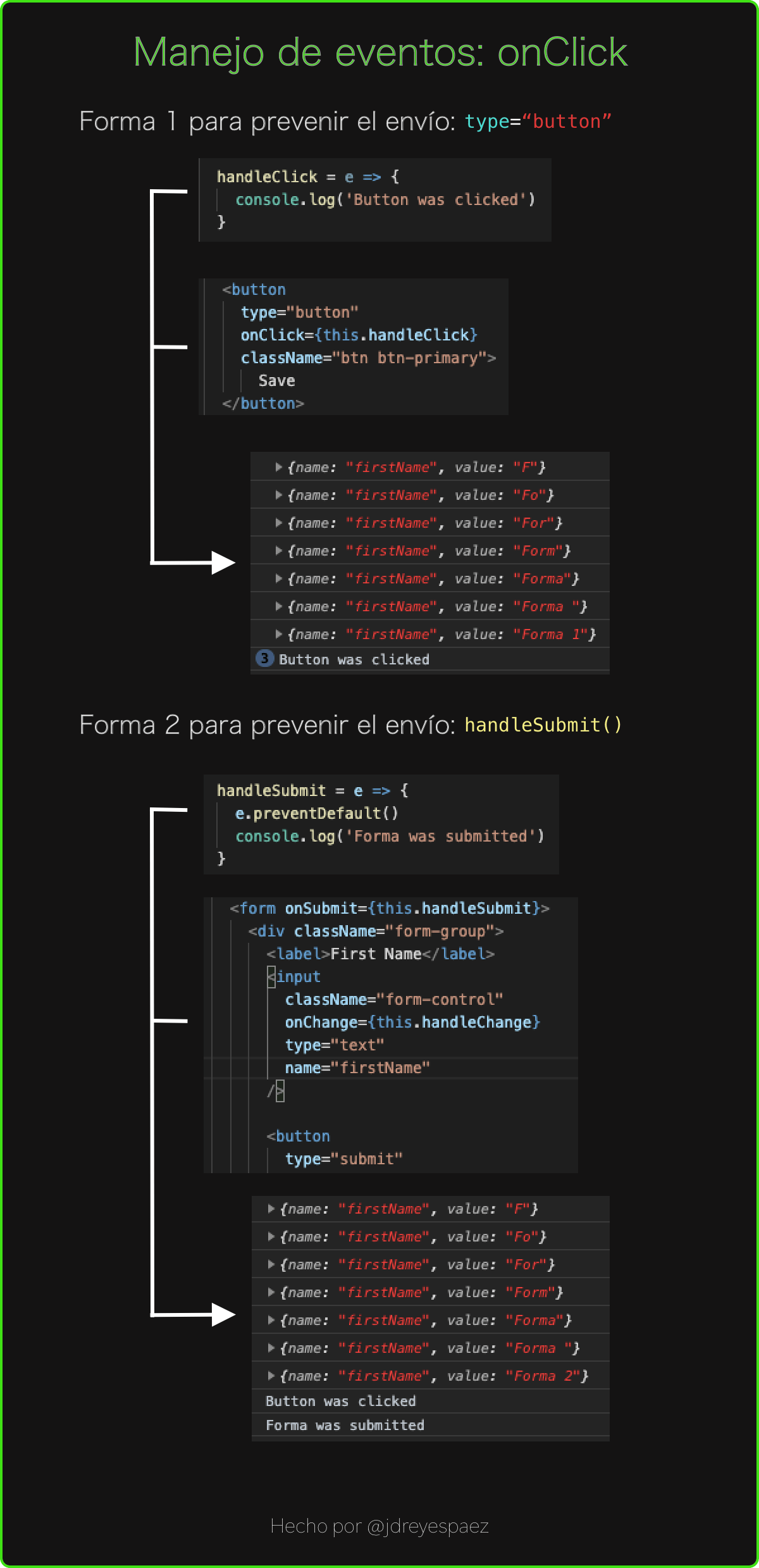
Podemos hacer **destructuring** para acceder fácilmente a las props.





**Enlazando eventos.**

* React dispone de **eventos**. Cada vez que se recibe información en un input se obtiene un evento **onChange** y se maneja con un método de la clase *this.handleChange*
* Los elementos *button* también tienen un evento que es **onClick**.
* Cuando hay un botón dentro de un formulario, este automáticamente será de tipo **submit**. Si no queremos que pase así hay dos maneras de evitarlo: especificando que su valor es de tipo *type=*”*button”* o manejándolo desde el formulario cuando ocurre el evento **onSubmit** *e.preventDefault();*



**Manejo de estado**

Hasta esta clase todos los componentes han obtenido su información a través de *props* que vienen desde afuera (otros componentes) pero hay otra manera en la que los componentes pueden producir su propia información y guardarla para ser consumida o pasada a otros componentes a través de sus *props*. La clave está en que la información del **state** a otros componentes pasará en una sola dirección y podrá ser consumida pero no modificada.

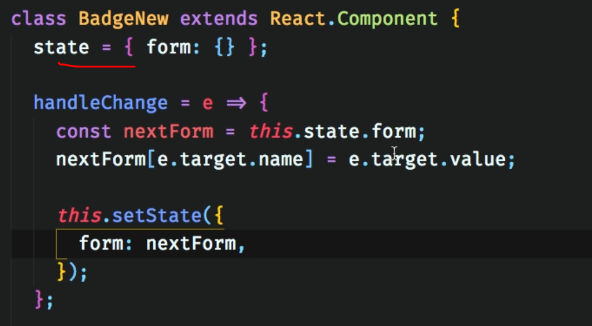
* Para guardar la información en el estado se usa una función de la clase *component* llamada **this.setState** a la cual se le debe pasar un objeto con la información que se quiere guardar.
* El **state** en React es un objeto que deriva de la clase React Component. Puede definirse a sí mismo como un objeto que almacena las propiedades observables que controlan el comportamiento del componente. (como en este caso el value de los inputs de los formularios)



* Aquí lo que estamos haciendo es que la información está siendo guardada en dos sitios. Cada *input* tiene una funcion enlazada a su evento OnChange así este guarda su propio valor y al tiempo la está guardando en *setState*, lo cual no es ideal. Para solucionarlo hay que modificar los *inputs* de un estado de no controlados a controlados. Para eso vamos a tener que agregar a nuestro input el atributo **Value**



Es importante tener en cuenta que este **estado** de nuestro componente al ser un objeto debe estar inicializado al principio de nuestro componente.

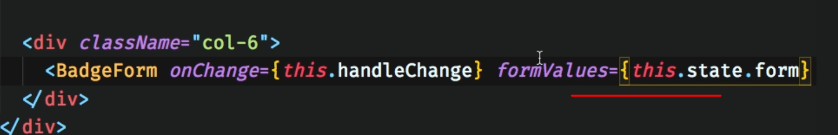


# Levantamiento del estado

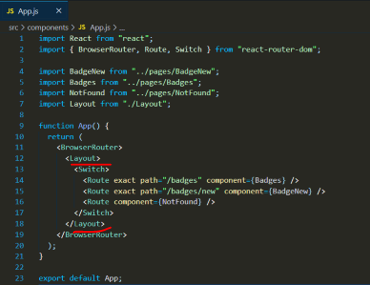
**Levantar el estado** es una técnica de React que pone el estado en una localización donde se le pueda pasar como props a los componentes. Lo ideal es poner el estado en el lugar más cercano a todos los componentes que quieren compartir esa información.

Algo interesante que le da el nombre a React es su parte de “reactivo” ya que cada vez que hay un cambio en el estado o en los props que recibe un componente se vuelve a renderizar todo el componente y todos sus descendientes.



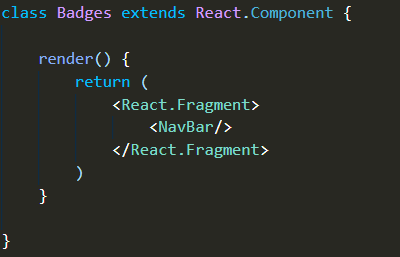


**Layouts.**

 Cuando tenemos elementos en nuestro DOM que se repiten podremos agregar toda esta configuración en un componente llamado App. En este incluiremos la configuración del React Router y agregaremos dentro, otro componente **Layout que será elementos que se repiten (como un NavBar).**

# React.Fragment.

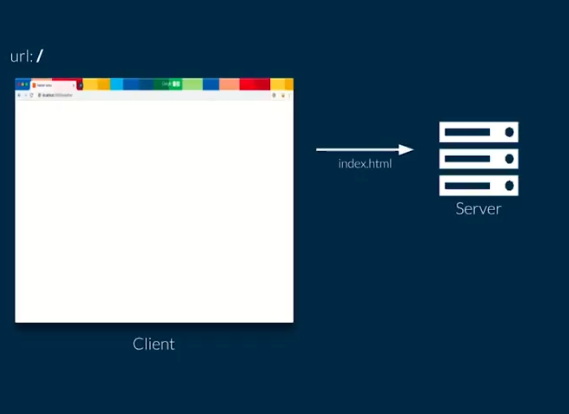
* React.Fragment es la herramienta que te ayudará a renderizar varios componentes y/o elementos sin necesidad de colocar un div o cualquier otro elemento de **HTML** para renderizar sus hijos. Al usar esta característica de React podremos renderizar un código más limpio y legible, ya que ``React.Fragment` no se renderiza en el navegador.



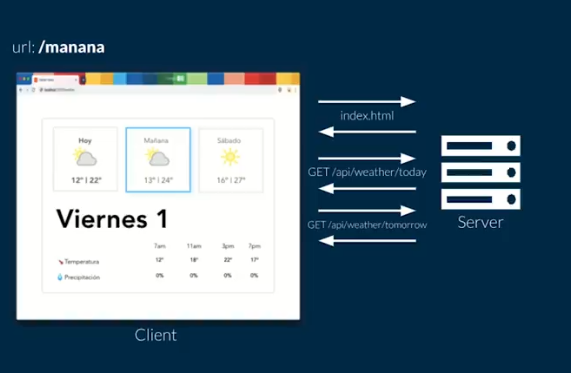
**Introducción a React Router**

Las aplicaciones que se trabajan en React son llamadas **single page apps**. Esto es posible gracias a **React Router** que es una librería *Open Source*.

**Multi Page Apps (MPA):** Cada página implica una petición al servidor. La respuesta usualmente tiene todo el contenido de la página, por lo que de esta manera era mucho más lento la visualización del contenido en nuestro navegador. Esta era la forma antigua.

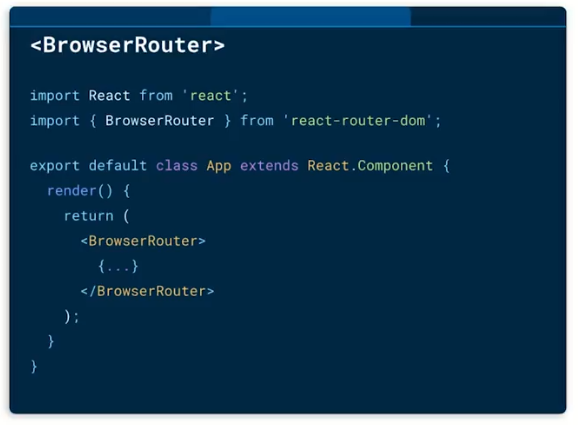


**Single Page Apps (SPA):** Aplicaciones que cargan una sola página de HTML y cualquier actualización la hacen reescribiendo el HTML que ya tenían. Esto es lo que se logra por ejemplo con JS y React.

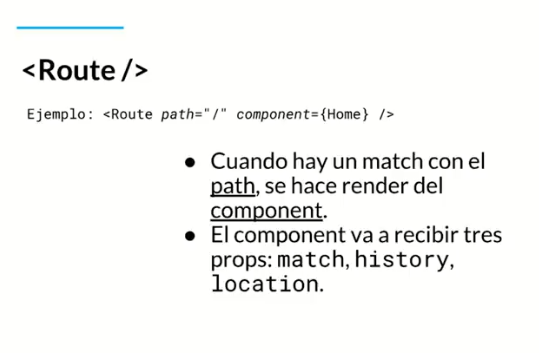


**React Router (v4):** Nos da las herramientas para poder hacer SPA fácilmente. Usaremos 4 componentes:

* **BrowserRouter**: es un componente que debe estar siempre lo más arriba de la aplicación. Todo lo que esté adentro funcionará como una SPA.

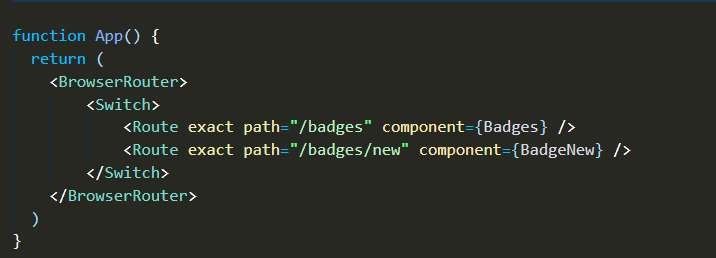


* **Route**: Cuando hay un match con el path, se hace render del component. El component va a recibir tres props: match, history, location.

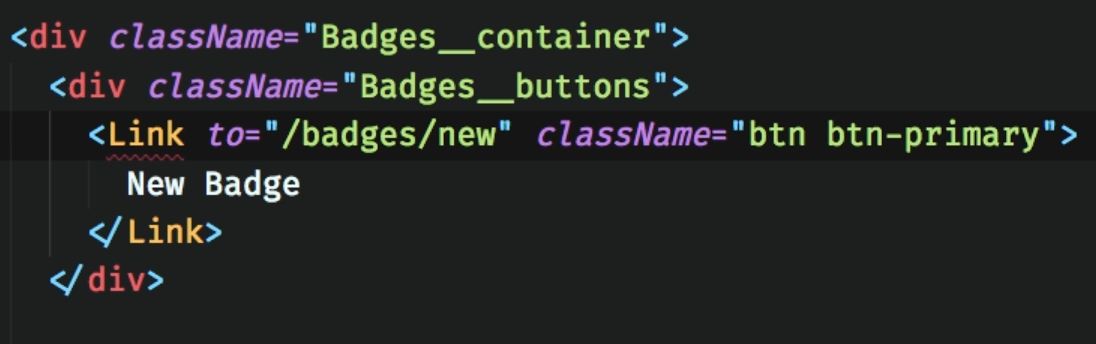


* **Switch**: Dentro de Switch solamente van elementos de Route. Switch se asegura que solamente un Route se renderice.
* **Link**: Toma el lugar del elemento <a>, evita que se recargue la página completamente y actualiza la URL.

Ejemplo:



Con el atributo **exact** vamos a decir que la ruta tiene que ser exactamente como se la configuramos, ya que si no colocáramos este atributo, en este caso cuando quisiéramos entrar a **/badge/new** entraría a **/badges** porque al identificar que coinciden en sus palabras me llevara a esa ruta por coincidencia.



Aquí lo que hicimos es que cambiamos el **<a>** por un etiqueta que nos ofrece react-router llamada **<Link>** con el que el atributo **to** indicaremos hacia donde va ir como **href.**



**Usar React Router.**

Para instalar **React Router** lo hacemos desde la terminal con:

*npm install react-router-dom*

Como es importante usar exactamente la misma versión, del package.json en “dependencies” se quita lo que está delante del 4.

* **Link** internamente tiene un elemento <a> pero va a interceptar el clic para navegar de manera interna sin refrescar toda la página.
* El 404 es la ruta que se renderizará cuando ninguna otra coincida con la dirección ingresada.



Otra forma de hacer que todas tus URL’s que no existan sean redirigidas a tu componente de 404 sería de la siguiente forma:

# 

Como podemos observar llamamos a nuestro componente 404 y luego utilizamos Redirect, el cual es un componente de **React Router** para hacer redirecciones; en este caso hacemos que todas las URL’s que no correspondan a alguna que hayamos declarado, sean redirigidas a MiComponente404.

# Ilustraciones en <https://stories.freepik.com/>

# 

**Introducción del ciclo de vida de un componente**

Cuando React renderiza los componentes decimos que entran en escena, cuando su estado cambia o recibe unos *props* diferentes se actualizan y cuando cambiamos de página se dice que se desmontan.

**Montaje:**

* Representa el momento donde se inserta el código del componente en el DOM.
* Se llaman tres métodos: *constructor*, *render*, *componentDidMount*.

**Actualización:**

* Ocurre cuando los *props* o el estado del componente cambian.
* Se llaman dos métodos: *render*, *componentDidUpdate*.

**Desmontaje:**

* Nos da la oportunidad de hacer limpieza de nuestro componente.
* Se llama un método: *componentWillUnmount*.

Ciclo de vida de un componente de React.docx

# 

# Ejemplo de un uso claro.

# En un caso muy específico en el que podemos usar componentWillUnmount() es en lel uso de limpiar nuestro componente ya que esta funcion es llamada antes de desmontarse.

# En este ejemplo podemos ver que hay un componente en el que guarda una propiedad interna llamada this.timeOutId en el que almacena el Id de una funcion setTimeout, para que eventualmente en caso de que esta funcion aun no responda y el componente ya no se encuentre en el DOM (por cuestiones de interacción del usuario) y no nos vaya a generar errores, con componentWillUnmount() va a hacer que ANTES nos cancele esta funcion en su proceso de Desmontaje y no genere errores.

# Para ello existe una función que nos permite limpiar estas funciones Timeout. clearTimeout()

# 

# 

**Introducción llamadas a un API**

Las llamadas a una API siguen un patrón similar siempre que las hacemos, cada llamada consta de tres estados:

* **Loading**: cuando la petición se envía y estamos esperando.
* **Error**: se debe dejar un mensaje para el usuario para arreglar el error o volver a intentarlo.
* **Data**: los datos nos pueden llegar de dos formas, o en error o con los datos requeridos.

# 

# Los datos nos pueden llegar vacíos o con datos.

# React.js: Cómo traer datos de un API en React

Una llamada a una **API** es un proceso asíncrono, es decir que lo comenzamos pero no sabemos cuándo acabará. Por lo mismo la función a escribir debe ser asíncrona.  
La llamada se hará usando **fetch** que es una función de JavaScript que al pasarle una dirección de internet, hará una petición **GET** y lo que sea que exista ahí será devuelto.

# 

# Controlar Visualmente el llamado a una API.

# Para ello, en la state del componente inicializamos unas propiedades con el que vamos a estar controlando según cambien de valor. Por ejemple el Loading de True a False.

# 

# 

# Según corresponda, estas propiedades en el state nos permitirán ir cambiando elementos en el DOM en nuestro render.

# 

# Otra manera de realizarlo es:

# 

# Skeleton como herramienta de Loader.

# <https://www.npmjs.com/package/react-loading-skeleton>

# <https://loading.io/css/>

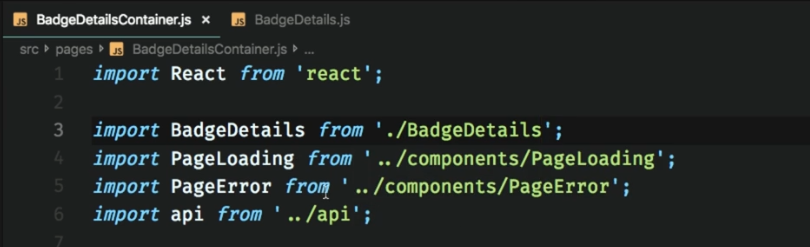
# Herramienta para barra de progreso.

# <https://react-bootstrap.github.io/components/progress/>

# UI Components y Container Components

En la programación es bueno separar las tareas en diferentes funciones y en React sucede lo mismo. Cuando un componente hace demasiado, probablemente es mejor dividirlo en dos.

Esta técnica de componentes presentacionales y componentes container es común, útil y hace parte de las buenas prácticas.



# Portales

Hay momentos en los que queremos renderizar un modal, un tooltip, etc. Esto puede volverse algo complicado ya sea por la presencia de un z-index o un overflow hidden.

En estos casos lo ideal será renderizar en un nodo completamente aparte y para esto React tiene una herramienta llamada **Portales** que funcionan parecido a ReactDOM.render; se les dice qué se desea renderizar y dónde, con la diferencia de que ese dónde puede ser fuera de la aplicación.

# 

# 

# Modales

La técnica de usar componentes genéricos para crear uno nuevo especializado se llama **composición** y es una herramienta que todo buen programador debe saber utilizar.

**La técnica de composición**

*- Crear específico en base a genérico, extraer instancias de un componente padre*

***Podemos revisar el ejemplo del Modal creado de este curso.***

**Hooks.**

Las funciones no tienen un estado propio que manejar como ciclos de vida a los que deben suscribirse, mientras tanto las clases sí cuentan con ello.

React tiene un feature llamado **Hooks** que permite que las funciones también tengan *features* que solamente tienen las clases.

**Hooks:** Permiten a los componentes funcionales tener características que solo las clases tienen:

* **useState:** Para manejo de estado.
* **useEffect:** Para suscribir el componente a su ciclo de vida.
* **useReducer:** Ejecutar un efecto basado en una acción.

**Custom Hooks:** Usamos los *hooks* fundamentales para crear nuevos *hooks custom*. Estos *hooks* irán en su propia función y su nombre comenzará con la palabra *use*. Otra de sus características es que no pueden ser ejecutados condicionalmente (*if*).

* *useState* regresa un arreglo de dos argumentos.

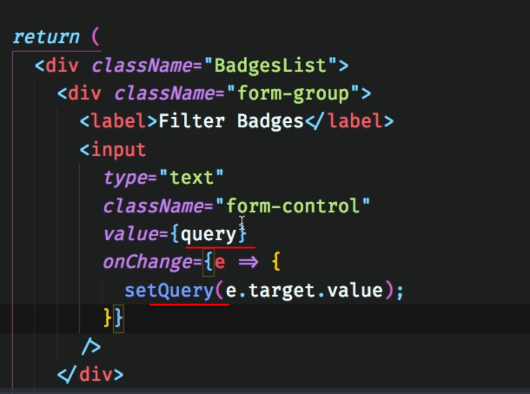
<https://desarrollofront.medium.com/entendiendo-los-hooks-de-react-c%C3%B3mo-usar-usestate-y-useeffect-en-nuestros-componentes-611b9e826dfa>

<https://es.reactjs.org/docs/hooks-custom.html>

En este caso podemos ver que a la funcion **BadgesList**  que es un componente funcional, vamos a agregarle un **state** como si fuera un componente de tipo clase con **React.useState.**



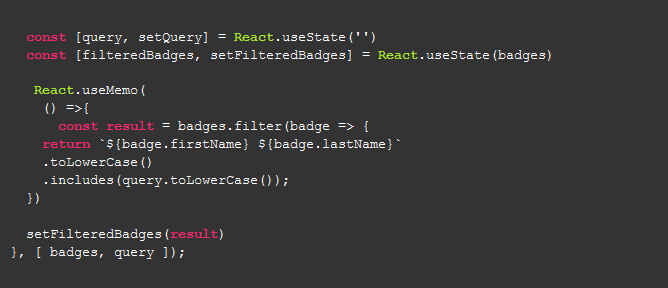
En este caso el **query** va a ser una propiedad del estado de este componente. Y **setQuery** nos va a permitircambiar el valor de este propiedad:



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

**useMemo.**

Otro hook que trae React que se llama useMemo, le vamos a dar un función y unos argumentos y la primera vez que reciba ese par de argumentos va a correr la función y va a calcular el resultado y lo regresa, pero la segunda vez que tenga esos argumentos de nuevo ya tiene la contestación memorizada y la va a poder regresar mucho más rápido.

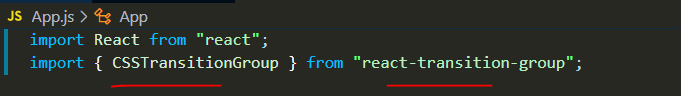


# Animaciones con React Transicion Group.

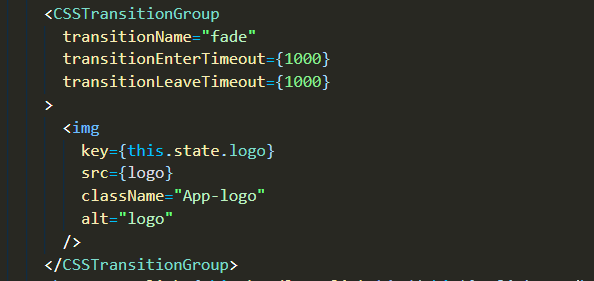
* Instalación:

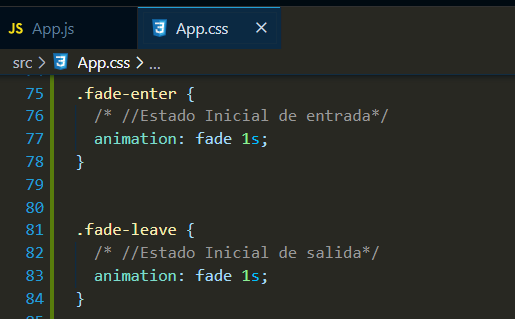
npm install react-transition-group --save

* Importar el componente **CSSTransitionGroup** de la siguiente forma:



* La sintaxis de la transición queda de la siguiente forma:





# Añadiendo Cheet JS.

## [Añadiendo Cheet JS](https://github.com/pablojorgeandres/notas-clases/blob/master/animacionesWeb.md#clase36)

[cheet.js](https://namuol.github.io/cheet.js/) es una librería que nos permite ‘escuchar’ el teclado de una manera muy sencilla.

**Instalar**:

project-folder npm install cheet.js

Cheet recibe dos parámetros para funcionar. El primero es el string o set de teclas del keyboard que queremos ‘oír’ y el segundo una función con lo que queremos que suceda si el string es tipeado por el cliente.

cheet( 's t r i n g', () => {} )

Usaremos cheet para animar nuestros llamados ‘easterEggs’ o animaciones sorpresa.

# Importants Questions and Answers.

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 