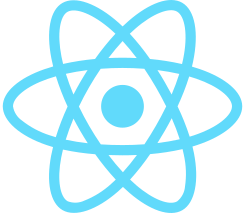
***Notas del Curso.***

**¿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**

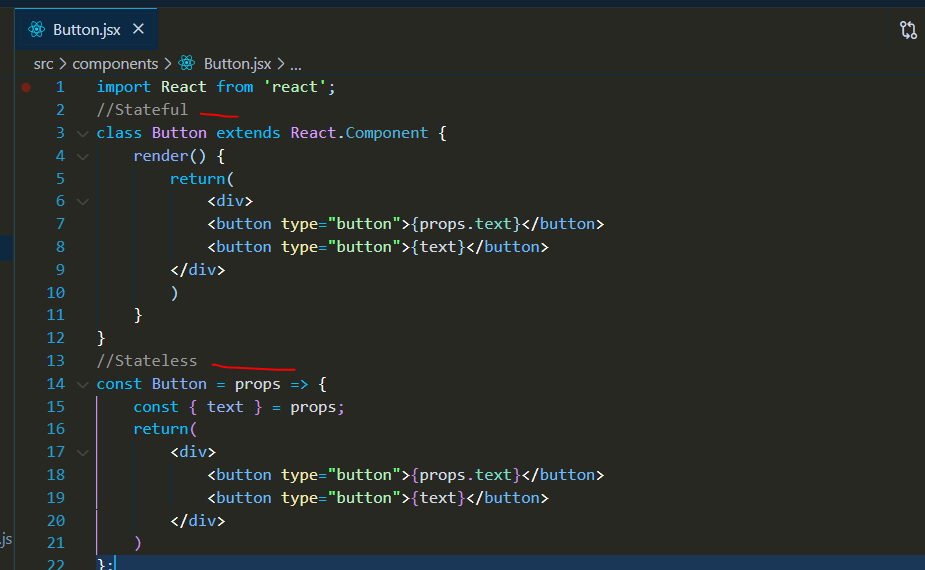
Los nombres de nuestros componentes deben empezar con una letra mayúscula, al igual que cada nueva palabra del componente. Esto lo conocemos como Pascal Case o Upper Camel Case.

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).



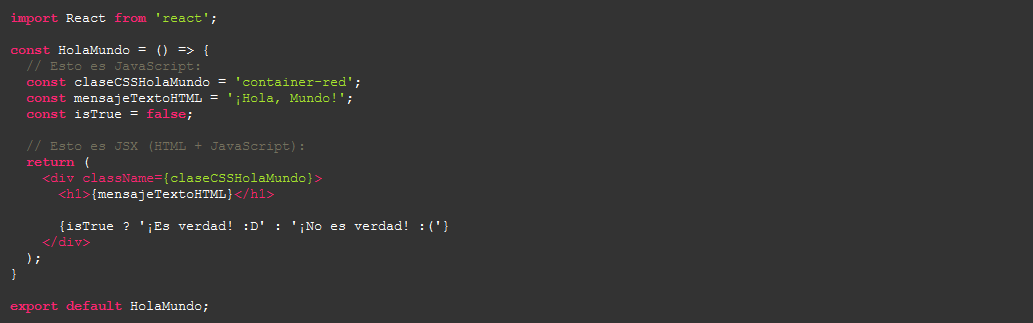
**Stateless VS Stateful.**



# JSX: JavaScript + HTML

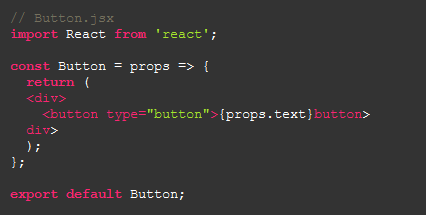
Estamos acostumbrados a escribir código HTML en archivos .html y la lógica de JavaScript en archivos .js.

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

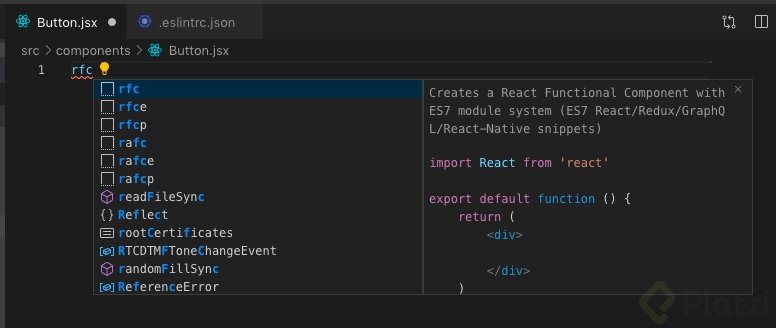
Las **Props** son la forma de enviar y recibir información en nuestros componentes. Son la forma de comunicar cada componente con el resto de la aplicación. Son muy parecidas a los parámetros y argumentos de las funciones en cualquier lenguaje de programación.



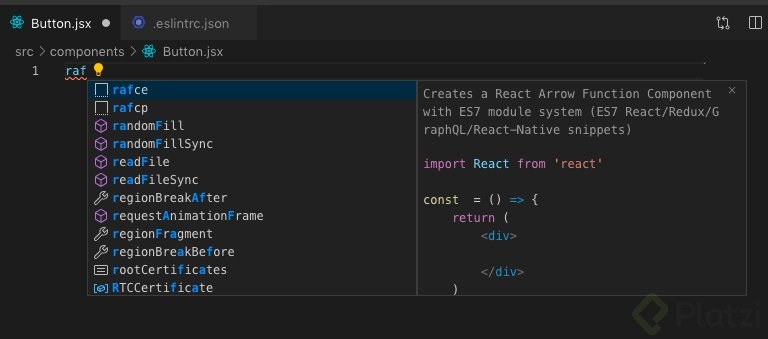


# Extension Útil





**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.

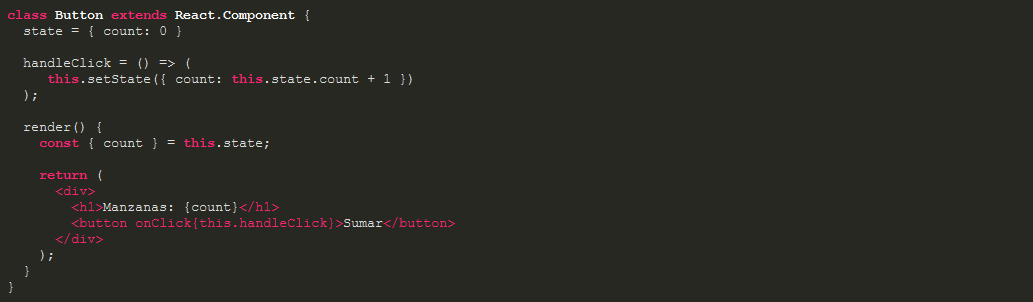
Ahora que entendemos cada una de las fases que tiene el ciclo de vida de react, podemos utilizarlas según sea necesario en nuestra aplicación y de esta forma crear las interacciones que necesitemos.

# 

# 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.

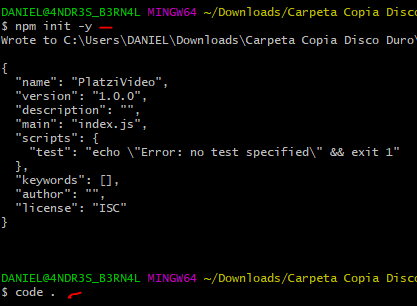
# Instalación y configuración de entorno.

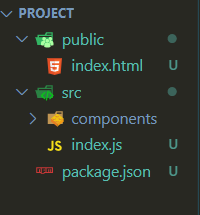
1. Iniciar un repositorio en GIT:



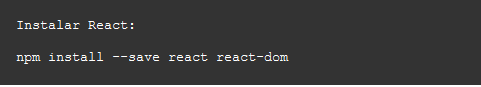
Iniciar un proyecto de Node.js:



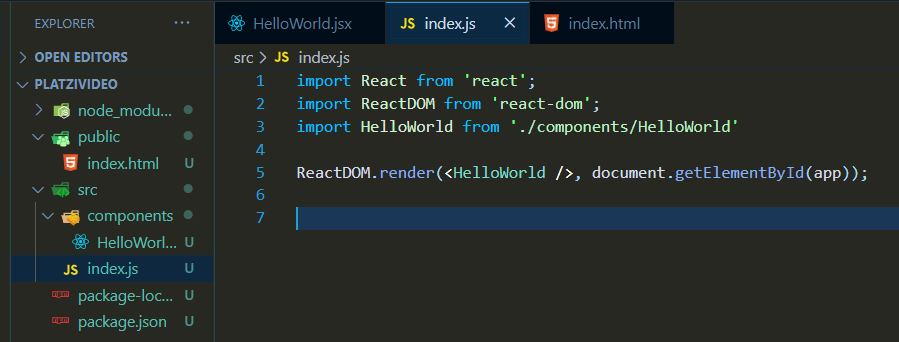


Configuramos nuestras carpetas que queden de la siguiente manera.

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



Usando React Inicialmente:

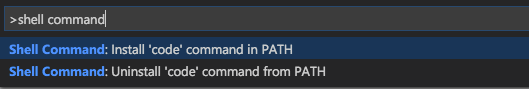




En caso de no funcionar:



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.

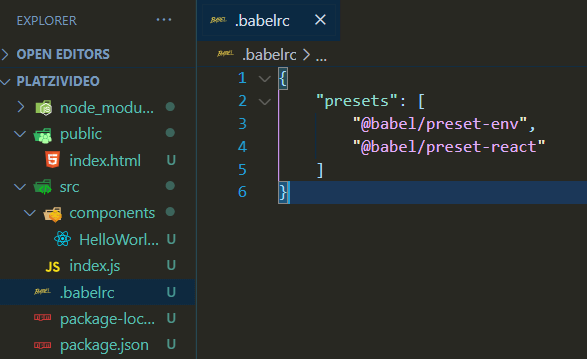
Instalación de Babel y otras herramientas para que funcione con React:

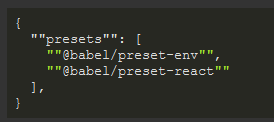


En caso previo de algún error durante el proceso:



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',

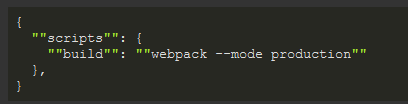
      filename: './index.html',

    }),

  ],

};

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

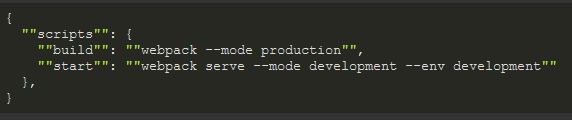


# Webpack Dev Server: Reporte de errores y Cambios en tiempo real

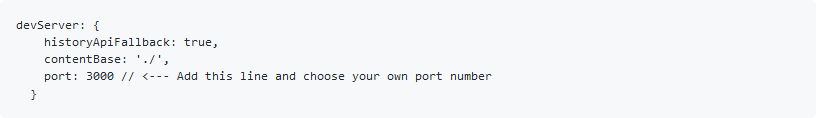
Instalación de **Webpack Dev Server**:



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:



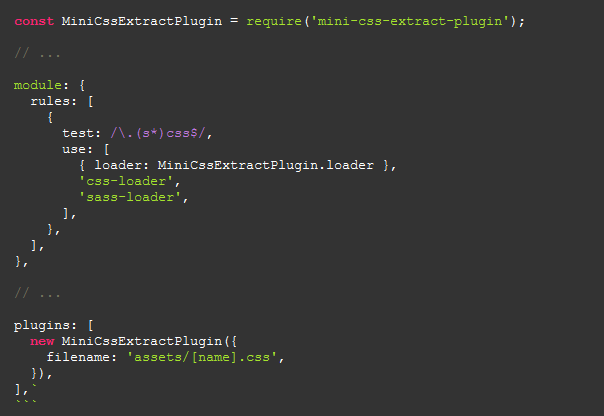
# Estilos con SASS

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.

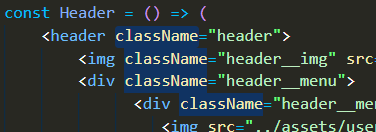
<https://gist.github.com/gndx/747a8913d12e96ff8374e2125efde544>

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.

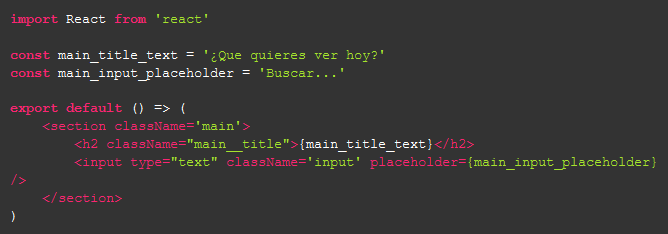
<https://gist.github.com/gndx/60ae8b1807263e3a55f790ed17c4c57a>

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

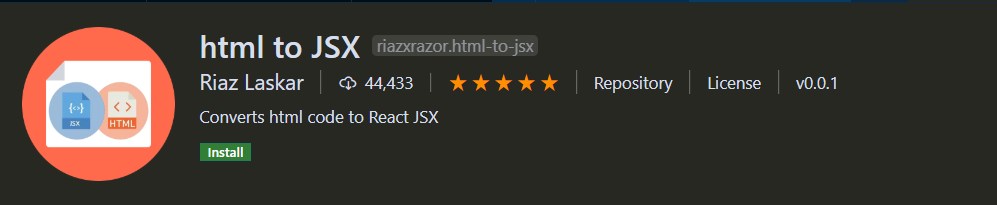
**Recomendación.**

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

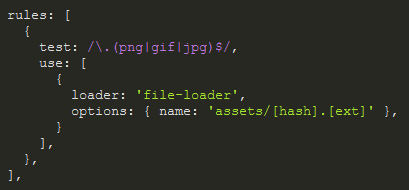
Vamos a usar **File Loader** para acceder a las imágenes de nuestro proyecto desde el código.

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.

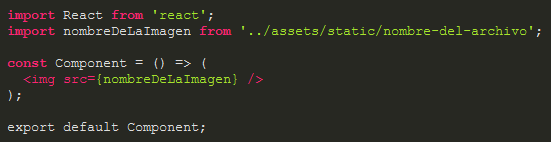
Instalación de File Loader:



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

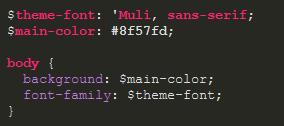


Uso de File Loader con React:

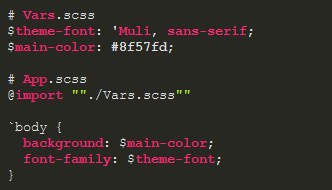


# Imports, Variables y Fuentes de Google en Sass

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



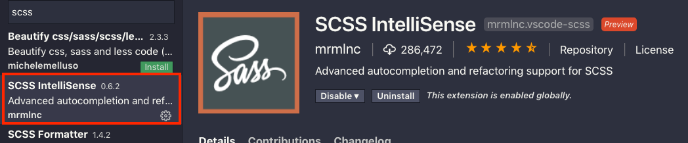
Podemos guardar nuestras variables en un archivo especial e importarlo desde los archivos de estilo donde queremos usar estas variables.



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

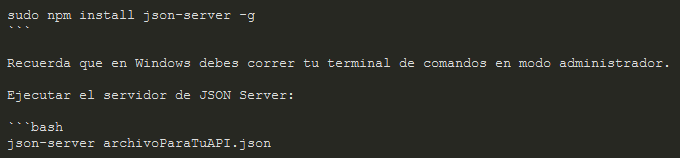
Vamos a usar **JSON Server** para crear una Fake API: una API ““falsa”” construida a partir de un archivo JSON que nos permite preparar nuestro código para consumir una API de verdad en el futuro.

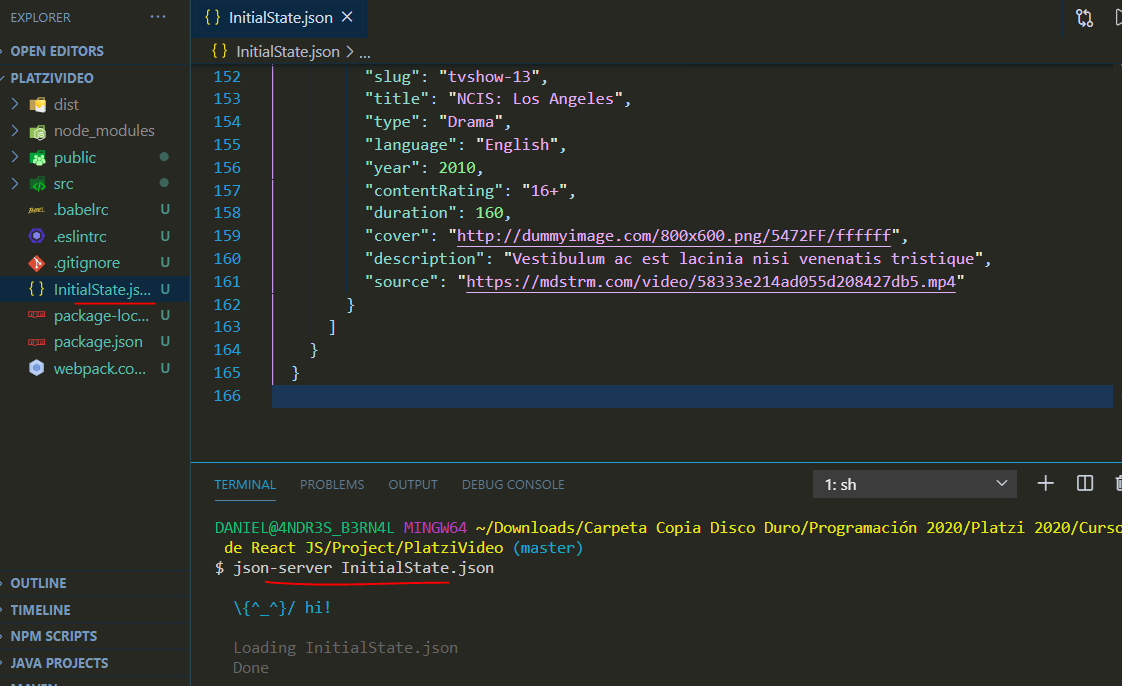
Este archivo lo llamamos **InitialState.js** en el que contendrá toda la información en JSON.

Ej:

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

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

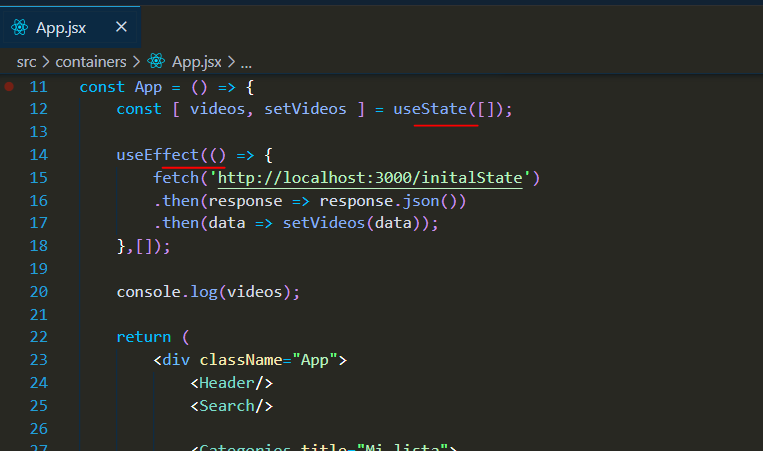




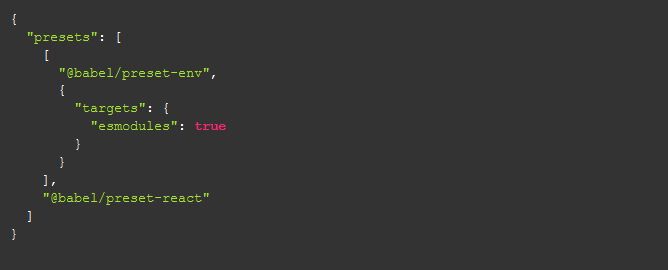
# React Hooks: useEffect y useState.

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.

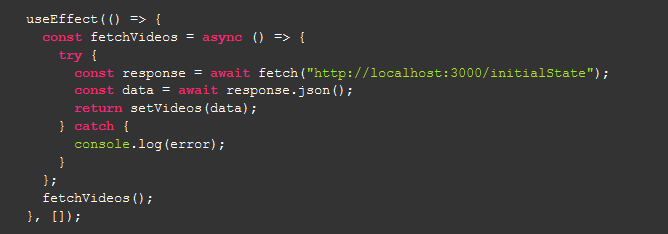
React es una librería desarrollada por Facebook que nos ayuda a construir interfaces de usuario interactivas para todo tipo de aplicaciones: páginas web, aplicaciones móviles o de escritorio, experiencias de realidad virtual, entre otras.



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).

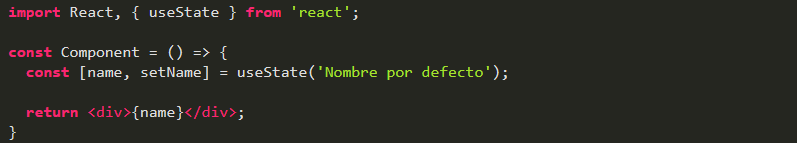
El hook **useEffect** es una funcion que nos permite ejecutar código cuando se monta, desmonta o actualiza un componente. El primer argumento que recibe es una funcion que se ejecutará cuando React monte o actualice el componente, esta funcion puede devolver otra funcion que se ejecuta cuando el componente 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.

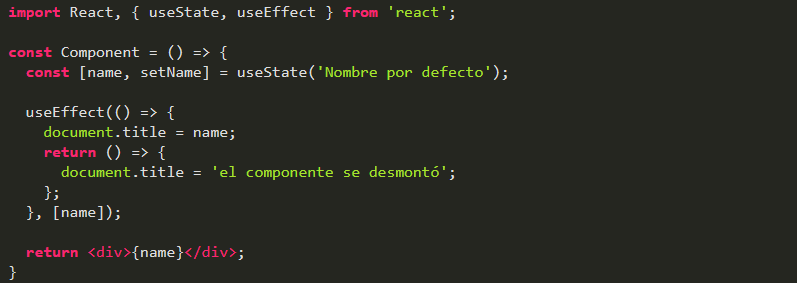
El argumento que enviamos a esta función es el valor por defecto de nuestro estado (initial state).

 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>

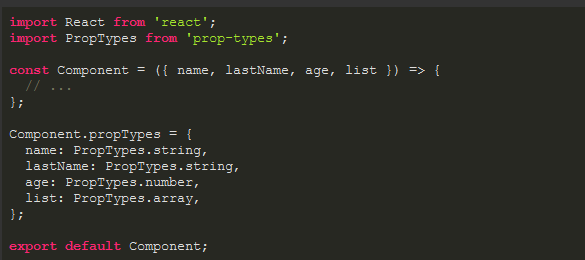
# PropTypes

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

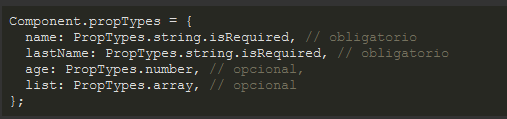
Instalación de PropTypes:



Uso de PropTypes:

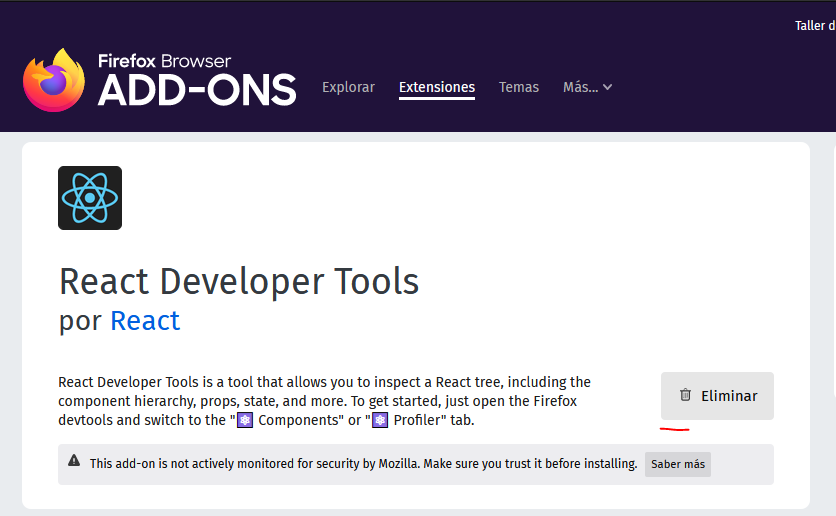


Por defecto, enviar todas nuestras props es opcional, pero con los propTypes podemos especificar cuáles props son obligatorias para que nuestro componente funcione correctamente con el atributo isRequired.



# 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.

