

INSTALACIÓN Y CONFIGURACIÓN DE NUESTRO ENTORNO DE DESARROLLO EN REACT

Primeros pasos:

Lo primero que tenemos que hacer es crear una carpeta con el comando **Mkdir** en la terminal de **Hyper Ubuntu** en **Linux** con el nombre de tu proyecto, asumo que eso ya lo sabes hacer, y luego hacemos lo siguiente. Como buena práctica escribimos en la terminal un **\$ git init** y luego **\$ npm init -y** (este último te crea un archivo **package.json**).

Abrimos el editor de código con el comando **code** .

Donde vamos a crear la carpeta **public** y su archivo **index.html** y la carpeta **src** con su archivo **index.js** (este es el más importante de todas las carpetas) porque es ahí donde se alojará todo nuestro código.

Luego procederemos a instalar **React** y como lo haremos con el siguiente comando

\$ npm install --save react react-dom

Después de que se haya instalado **React** nos daremos cuenta en el editor o en la carpeta de proyecto que se agregó un **package-lock.json** que permite manejar el versionado de nuestros paquetes y dependencias que vamos a instalar, después de eso vamos a crear las siguientes carpetas en **src**

components (en esta carpeta haremos nuestros componentes de React)

pages (esta carpeta nos va a servir mucho para manejar React Router y hacer un SPA (Single Page Application en React) aquí es donde se alojarán nuestras páginas)

assets: dentro de assets vamos a crear otra carpeta que se llamará **styles** (en styles se alojaran nuestros estilos Css y Sass).

Y listo ahora vamos a instalar las dependencias y paquetes que necesitará nuestro proyecto

Para seguir configurando nuestro entorno de desarrollo con React hacemos lo siguiente, instalar Babel que es una herramienta muy popular para escribir JavaScript moderno y transformarlo en código que pueda entender cualquier navegador.

Este es el comando que vamos a utilizar en la terminal para que instalemos Babel Correctamente:

```
$ npm install --save-dev @babel/core @babel/preset-env  
@babel/preset-react babel-loader
```

después de se instale vamos a crear un archivo en nuestra carpeta de proyecto que se llamará **.babelrc** tal y como lo ves escrito y luego copiamos y pegamos este siguiente código que nos ayudará a configurar Babel:

```
{  
  "presets": [  
    "@babel/preset-env",  
    "@babel/preset-react"  
  ]  
}
```

Bien, ahora vamos a codificar los archivos que están en las carpetas que hemos creado, es decir el index.html, el index.js y los componentes que vayamos a crear, todo con la finalidad de que nuestro entorno de desarrollo este totalmente configurado para poder trabajar cómodamente en React, te dejare imágenes para que sigas los pasos y lo hagas en tu editor.

Index.html

```
index.html X
public > index.html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Platzi Video</title>
7  </head>
8  <body>
9    <div id='app'></div>
10 </body>
11 </html>
```

eso es únicamente lo que tenemos que escribir en el index.html así que hazlo

Index.js

```
JS index.js X
src > JS index.js
1  import React from 'react'
2  import ReactDOM from 'react-dom'
3  import HelloWorld from './components/HelloWord'
4
5
6  ReactDOM.render(<HelloWord />, document.getElementById('app'))]
```

Eso tenemos que escribir en **index.js** no es nada más que eso, si te das cuenta hay una ruta que estamos trayendo desde **components** y es **HelloWord** que es un componente de **React** que yo he creado por aparte para luego llamarlo en **ReactDOM.render()**, eso lo veras en la siguiente imagen para que vea que código contiene ese archivo

HelloWord (un archivo de components)

```
JS HelloWorld.jsx X
src > components > JS HelloWorld.jsx
1  import React from 'react'
2
3
4  const HelloWorld = () => (
5    <h1>Hola mundo</h1>
6  )
7
8
9
10 export default HelloWorld
11
12
13
14
15
16
```

y este es el archivo **HelloWord.jsx** o con extensión **js** como quieras en este caso lo escribí con **jsx** que hace referencia a React, recuerda que React es JavaScript por eso podrás trabajar con cualquiera de las dos extensiones, si nos damos cuenta vemos un **hola mundo** en las etiquetas **<h1>** y es así la forma en la que se trabaja los componentes que nosotros hagamos, me cabe resaltar que hay dos métodos del ciclo de vida de React uno son los **class** y el otro los **hooks(funciones)** pero no me voy a extender tanto en eso porque para eso se hacen los cursos de React y así puedas entender a profundidad esta maravillosa librería de código abierto, así que sin más sigamos con las configuraciones de entorno que es lo más importante.

Bien, después de haber hecho todo este proceso vamos a instalar **webpack** de manera correcta y todo lo necesario para los proyectos que nosotros hagamos en un futuro, te sugiero que aprendas mucho de webpack para que entiendas muy bien su funcionamiento, para eso hay muchos cursos que puedes tomar

Pero que es webpack tal vez eso te preguntes, webpack nos sirve para compilar múltiples archivos como (JavaScript, Html, Css, imágenes, entre muchos otros) en uno solo. Es importante para que tu entorno de desarrollo quede listo, compiles tus archivos y lleves tu código a producción sin problemas. Esto va a optimizar tu trabajo.

la instalación de esta herramienta es la siguiente:

\$ npm install webpack webpack-cli html-webpack-plugin html-loader --save-dev

Después de que ya este instalado vamos a crear un nuevo archivo dentro de nuestro proyecto con este nombre **webpack.config.js** donde escribiremos el código de webpack para configurarlo, te recomiendo que copies y pegues este código para que no lo estés escribiendo

```
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'
  }),
]
};
```


Después de copiar y pegar todo ese código nos dirigimos al archivo de **package.json** y en **scripts** escribimos lo siguiente

```
"scripts": {  
  "build": "webpack --mode production"  
}
```

Y listo después de eso lo ejecutamos en la terminal de **Hyper Ubuntu** de la siguiente manera con el comando **\$ npm run build** y te tiene que aparecer compilado de esta manera para nosotros verificar que corre muy bien al ejecutar el comando

```
sergio@DESKTOP-VUQFGF0:/mnt/c/Users/sirja/platziVideo$ npm run build  
  
> platziVideo@1.0.0 build /mnt/c/Users/sirja/platziVideo  
> webpack --mode production  
  
Hash: de2ed0b9fb2ef3fde16d  
Version: webpack 4.44.1  
Time: 41103ms  
Built at: 08/13/2020 8:47:27 PM  


| Asset           | Size      | Chunks      | Chunk Names |
|-----------------|-----------|-------------|-------------|
| ./index.html    | 279 bytes | [emitted]   |             |
| assets/main.css | 49 bytes  | 0 [emitted] | main        |
| bundle.js       | 128 KiB   | 0 [emitted] | main        |

  
Entrypoint main = assets/main.css bundle.js  
[7] ./src/assets/styles/App.scss 39 bytes {0} [built]  
[8] ./src/index.js + 1 modules 425 bytes {0} [built]  
|   ./src/index.js 209 bytes [built]  
|   + 1 hidden module  
+ 8 hidden modules  
Child HtmlWebpackCompiler:  
  1 asset  
  Entrypoint HtmlWebpackPlugin_0 = __child-HtmlWebpackPlugin_0  
  [0] ./node_modules/html-webpack-plugin/lib/loader.js!./public/index.html 262 bytes {0} [built]  
Child mini-css-extract-plugin node_modules/css-loader/dist/cjs.js!node_modules/sass-loader/dist/cjs.js!src/assets/styles/App.scss:  
  Entrypoint mini-css-extract-plugin = *  
  [1] ./node_modules/css-loader/dist/cjs.js!./node_modules/sass-loader/dist/cjs.js!./src/assets/styles/App.scss 334 bytes {0} [built]  
  + 1 hidden module  
sergio@DESKTOP-VUQFGF0:/mnt/c/Users/sirja/platziVideo$
```

Ahora uno de los **recursos mas importantes** a la hora de trabajar con **React** es probar lo que estamos construyendo, para esto nos vamos a ayudar de un paquete de webpack y construir un **entorno de desarrollo local** que nos permita ver los cambios en tiempo real, para esto nos vamos a mover a nuestra terminal de comandos en **Hyper Ubuntu** e instalamos **Webpack Dev Server** de la siguiente manera:

```
$ npm install --save-dev webpack-dev-server
```

Después de que se haya instalado vamos al **package.json** y añadimos lo siguiente en **scripts**, es decir lo que te remarco en color azul

```
{  
  "scripts": {  
    "build": "webpack --mode production",  
    "start": "webpack-dev-server --open --mode development"  
  },  
}
```

Luego lo ejecutamos en la terminal con el comando **npm run start** y eso nos tiene que abrir un **localhost** con su respectivo **puerto** en el **navegador** y es ahí donde podremos visualizar todo el proyecto que estemos construyendo desde cero, junto con los cambios que nosotros hagamos en el código.

Bien vamos a instalar **Sass** con **npm** y luego lo configuraremos en **webpack.config.js** para que trabajemos con los estilos de nuestro proyecto a realizar, Sass es un preprocesador que te permite trabajar mucho más rápido en la creación de código CSS de manera eficiente y más dinámica por así decirlo, Gracias a Sass podemos escribir CSS con **variables**, **mixins**, **bucles**, entre otras características.

El comando correcto que utilizaremos en nuestra terminal es el siguiente:

```
$ npm install --save-dev mini-css-extract-plugin css-loader node-sass sass-loader
```

Después de que se instale, vamos a configurarlo en webpack y el código tiene que estar así para que funcione y lo que te resaltaré en color azul es lo que añadimos para configurarlo:

```
const path = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-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"
        }
      ]
    },
    {
      test: /\.?(s*)css$/,
      use: [
        {
          loader: MiniCssExtractPlugin.loader
        },
        'css-loader',
        'sass-loader'
      ],
    }
  ]
}
```

```

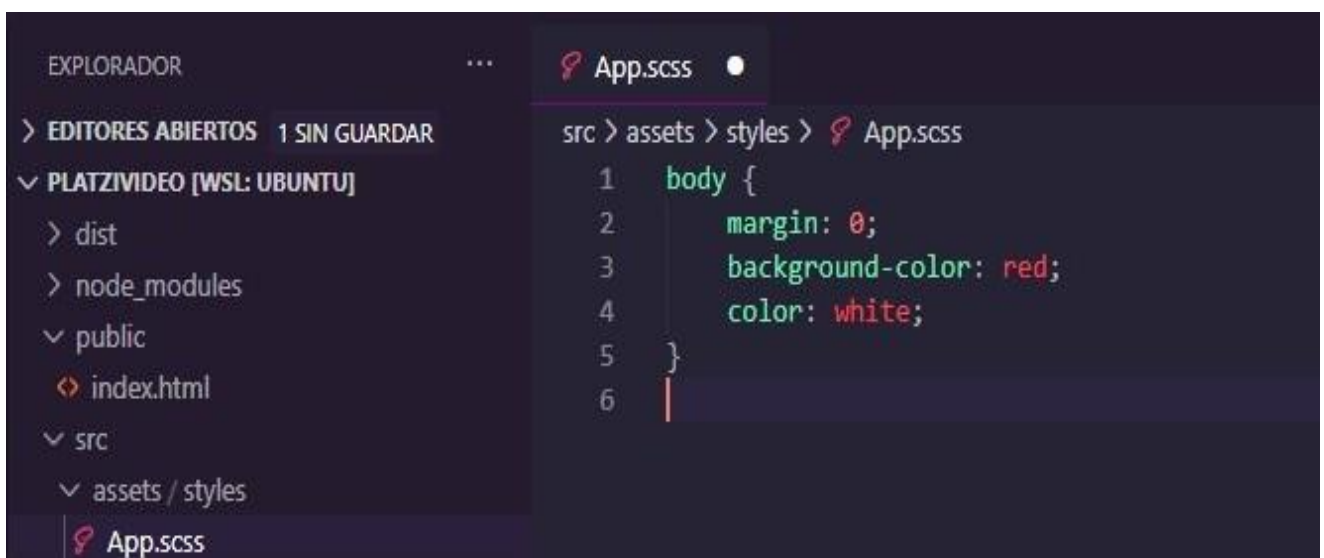
},
plugins: [
  new HtmlWebpackPlugin({
    template: './public/index.html',
    filename: './index.html'

  }),

  new MiniCssExtractPlugin({
    filename: 'assets/[name].css'
  }),
]
}

```

Y listo probamos con crear un archivo de **Sass**, cualquiera el que gustes hacer, para verificar que todo funciona correctamente, en este caso yo hice el archivo **App.scss** y escribimos un pequeño código CSS para que al hacer **npm run start** nos compile los estilos en el navegador te dejaré un pequeño ejemplo con una imagen, recuerda que los estilos siempre van en la carpeta de **assets**



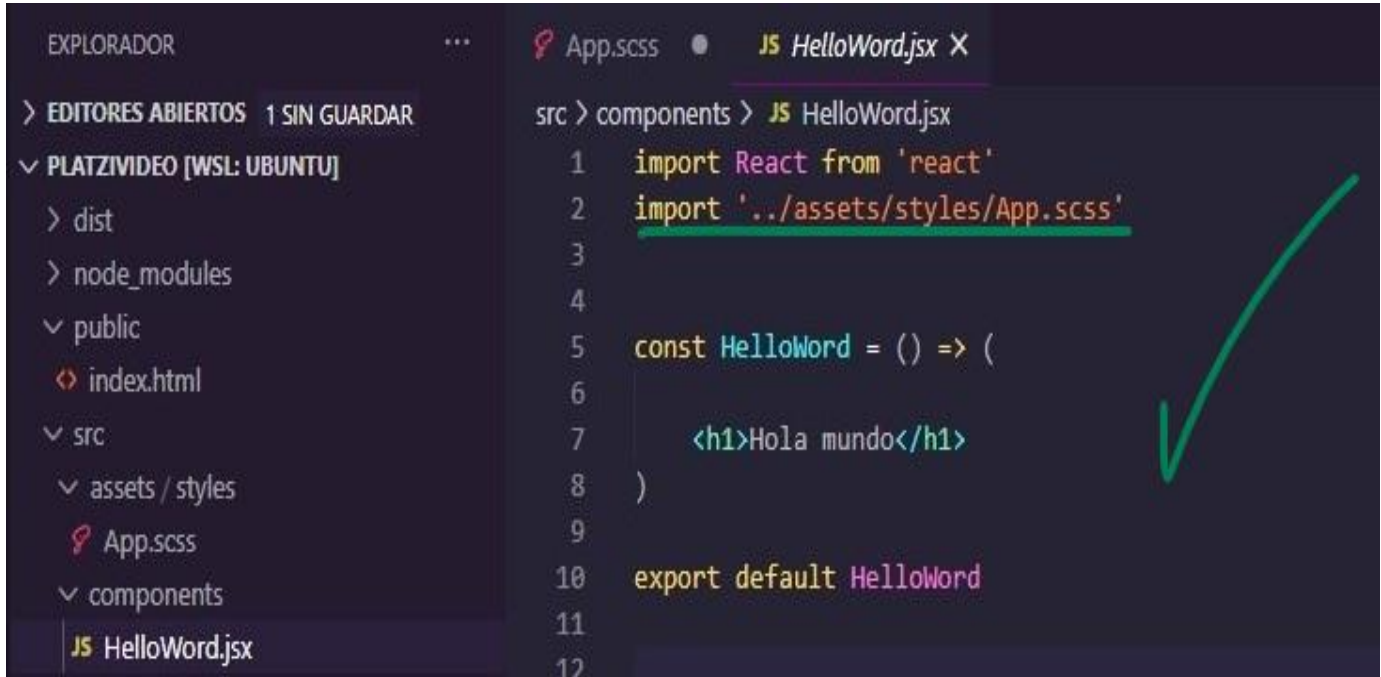
The screenshot shows a code editor with a dark theme. On the left, the 'EXPLORADOR' (Explorer) sidebar displays the project structure: 'EDITORIOS ABIERTOS 1 SIN GUARDAR', 'PLATZIVIDEO [WSL: UBUNTU]', 'dist', 'node_modules', 'public' (expanded), 'index.html', 'src' (expanded), and 'assets / styles' (expanded). The 'App.scss' file is selected under 'assets / styles'. The main editor area shows the content of 'App.scss' with the following code:

```

src > assets > styles > App.scss
1  body {
2    margin: 0;
3    background-color: red;
4    color: white;
5  }
6

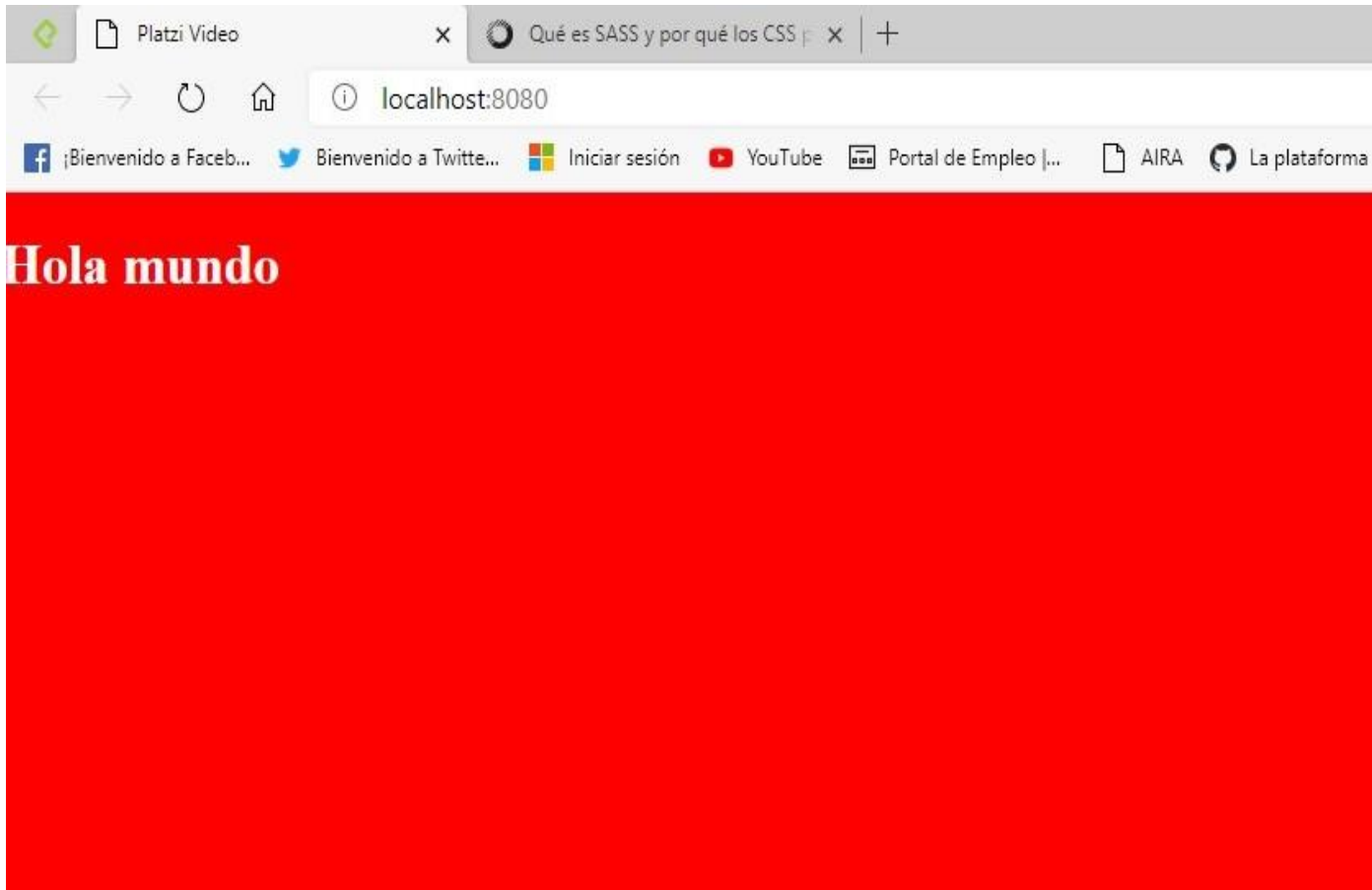
```

Bien, después de haber hecho ese pequeño código con Sass, **importamos** el archivo en **HelloWord.jsx** para que nos aparezca un **Hola mundo** como también el fondo en color **rojo** cuando estemos navegando y será de esta manera:



```
EXPLORADOR
...
App.scss JS HelloWord.jsx X
src > components > JS HelloWord.jsx
1 import React from 'react'
2 import '../assets/styles/App.scss'
3
4
5 const HelloWorld = () => (
6
7   <h1>Hola mundo</h1>
8 )
9
10 export default HelloWorld
11
12
```

Bien, ahora nos movemos a la terminal y con **npm run start**, probamos que todo está totalmente configurado y que, si funciona, deberías de visualizar esto en el navegador



Añadiendo imágenes con Webpack

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

Vamos a utilizar **webpack** para importar estos **archivos multimedia** que vamos a requerir en un futuro y también utilizando webpack los podremos separar en una sola carpeta para que estén listos en producción, actualizando nuestros componentes (o donde sea que usemos las imágenes) con los nuevos nombres y rutas de los archivos.

Instalación de File Loader:

```
$ npm install --save-dev file-loader
```

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

```
rules: [  
  {  
    test: /\.png|gif|jpg$/,  
    use: [  
      {  
        loader: 'file-loader',  
        options: { name: 'assets/[hash].[ext]' },  
      },  
    ],  
  },  
],
```


Uso de **File Loader** con React:

```
import React from 'react';  
import nombreDeLaImagen from '../assets/static/nombre-del-archivo.jpg';  
  
const Component = () => (  
  <img src={nombreDeLaImagen} />  
);  
  
export default Component;
```

nota: en la carpeta **assets** que es donde gestionamos nuestros estilos creamos una nueva carpeta con el nombre que tú quieras, por ejemplo, **estático** o **static** y es ahí donde almacenamos todas nuestras imágenes que reciban extensión **png**, **jpg**, **gif** hazlo para que funcione.

ULTIMAS CONFIGURACIONES PARA TU ENTORNO DE DESARROLLO

Nunca pero nunca olvidemos agregar a nuestra carpeta el **.gitignore** y ¿Por qué? ya te lo explicaré

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.

Luego instalamos y configuramos los **linters** como **ESLint** que 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.

Para instalarlo nos movemos a la terminal y escribimos el siguiente comando a continuación:

```
$ npm install --save-dev eslint babel-eslint eslint-config-airbnb  
eslint-plugin-import eslint-plugin-react eslint-plugin-jsx-a11y
```

Y cuando se haya instalado todo, debemos agregar un archivo en nuestro proyecto que se llame **.eslintrc** tal y como lo ves escrito y agregaremos todo el código **Json** que te dejaré en este link para que esté totalmente configurado

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

y también te dejaré este otro link para que copies y pegues todo en el archivo **git ignore**

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

