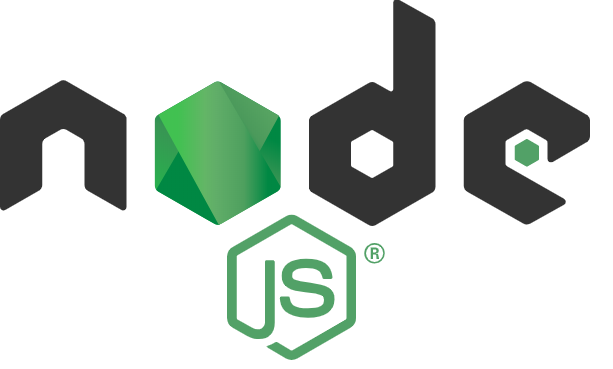
**CONFIGURACIÓN DE PROYECTO CON WEBPACK Y BABEL**

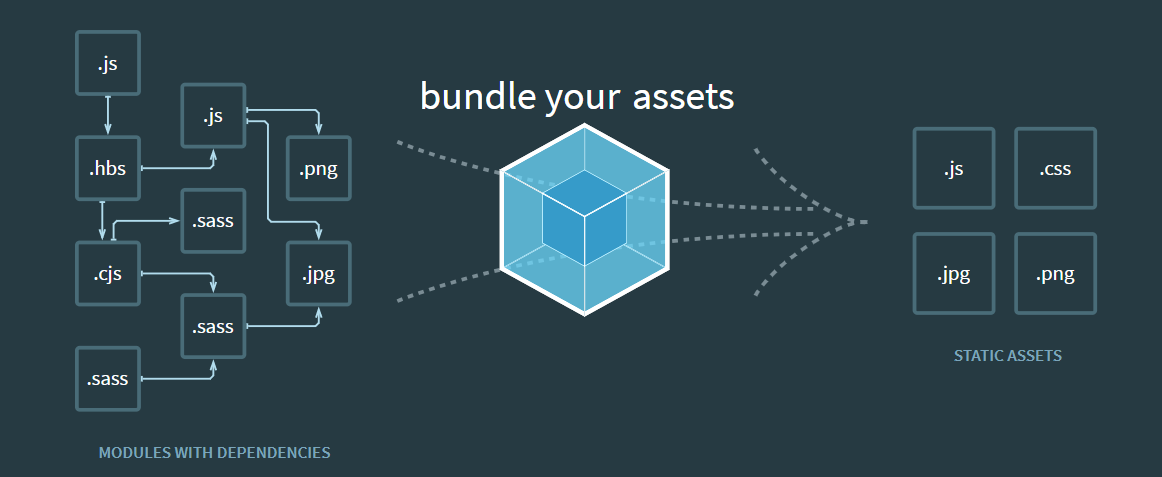
**Node.js:** es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Con relación a WebPack y Babel, Node js es fundamental para el correcto funcionamiento de las dependencias y demás librerías que son necesarias para tener proyectos configurados bajos estas tecnologías



**NPM:** Node Package Manager o simplemente npm es un gestor de paquetes, el cual hará más fáciles nuestras vidas al momento de trabajar con Node, ya que gracias a él podremos tener cualquier librería disponible con solo una línea de código, npm nos ayudará a administrar nuestros módulos, distribuir paquetes y agregar dependencias de una manera sencilla.



**WebPack:** En esencia, webpack es un paquete de módulos estáticos para aplicaciones JavaScript modernas. Cuando webpack procesa su aplicación, internamente crea un gráfico de dependencia que asigna cada módulo que su proyecto necesita y genera uno o más paquetes.

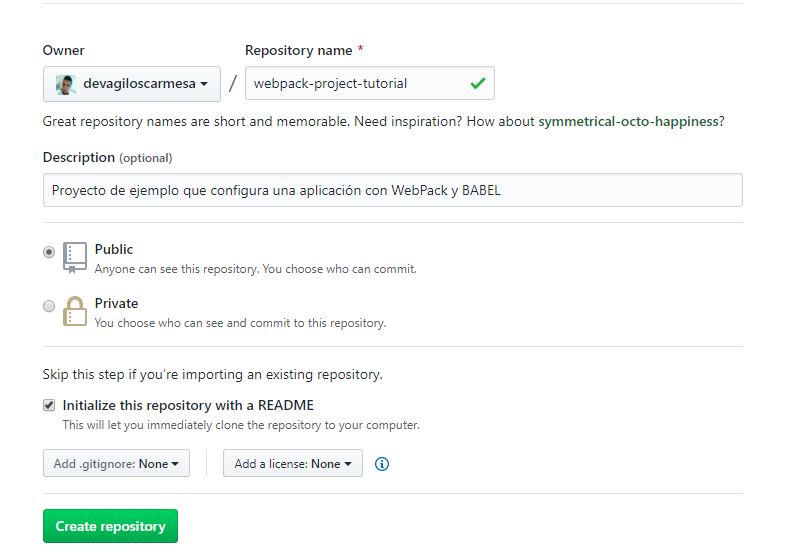


**BABEL:** Babel es una cadena de herramientas que se utiliza principalmente para convertir el código ECMAScript 2015+ en una versión de JavaScript compatible con versiones anteriores en navegadores o entornos antiguos.

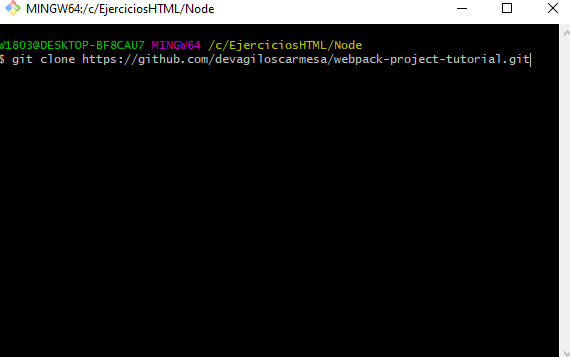


A continuación, veamos un ejemplo práctico donde se apliquen los conceptos de webpack y babel.

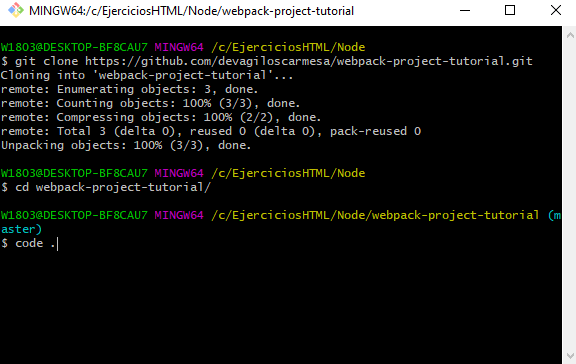
1. Crear un proyecto para tenerlo versionado (GitHub) preferiblemente (Si desean pueden crear solo el proyecto y abrirlo directamente con VSC que se encuentra en el paso # 3).



1. Se debe clonar el proyecto en algún directorio

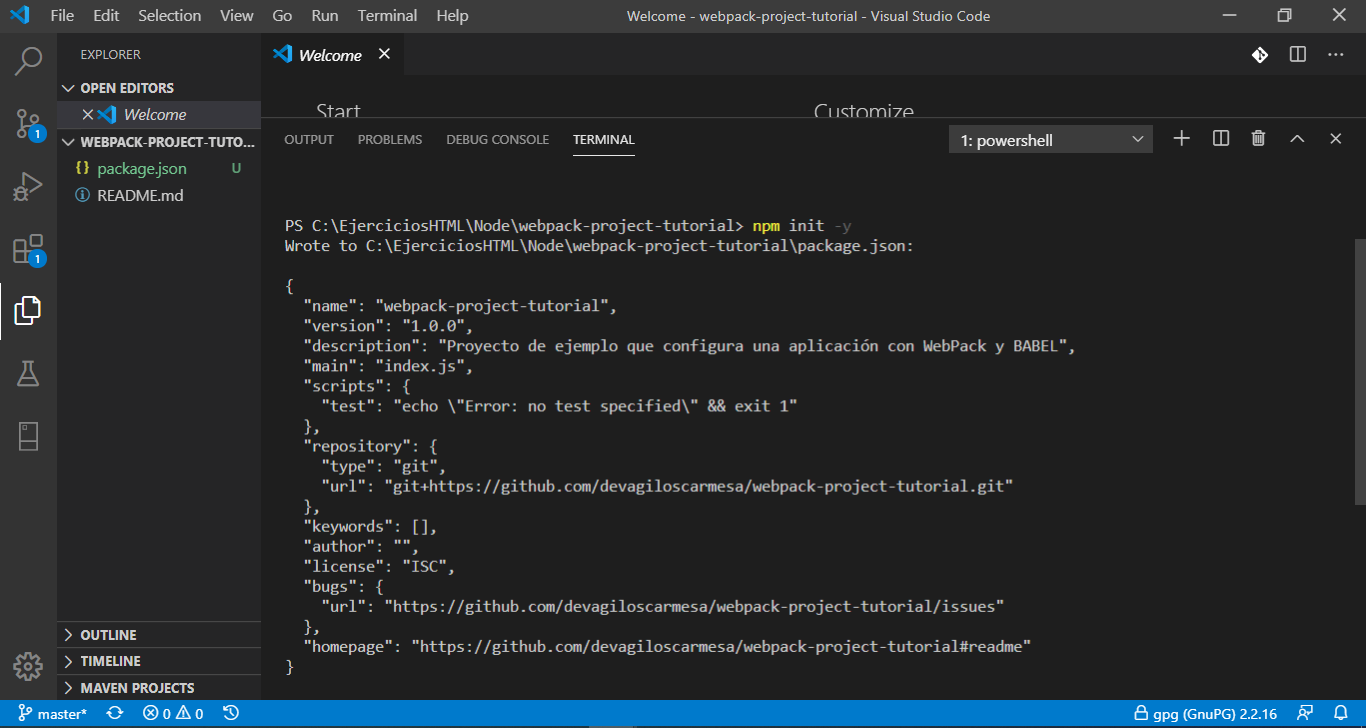


1. Abrimos el proyecto con VSC



1. Procedemos a crear un proyecto NPM con el fin de instalar las dependencias necesarias para que funcione webpack y babel.

npm init -y



1. Crear la siguiente estructura de archivos y carpetas, los cuales servirán para configurar *webpack* y *babel*.

/src

/src/app

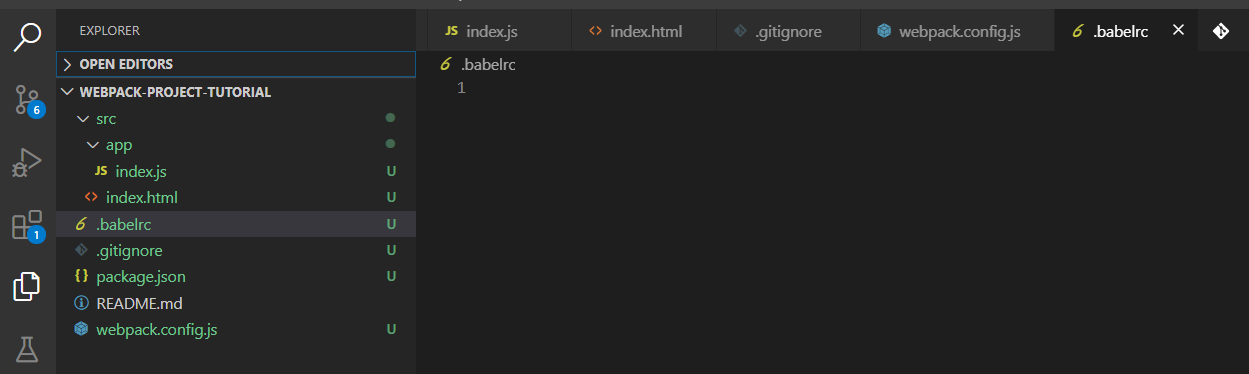
/src/index.html

/src/app/index.js

/.babelrc

/.gitignore

/webpack.config.js



1. Instalamos los siguientes módulos de mediante *NPM*.

yarn add webpack webpack-cli @babel/core @babel/cli @babel/preset-env @babel/polyfill babel-loader html-webpack-plugin webpack-dev-server

[**webpack**](https://www.npmjs.com/package/webpack)**:** webpack es un paquete de módulos. Su objetivo principal es agrupar archivos JavaScript para su uso en un navegador, pero también es capaz de transformar, agrupar o empaquetar casi cualquier recurso o activo.

[**webpack-cli**](https://www.npmjs.com/package/webpack-cli): webpack CLI proporciona un conjunto flexible de comandos para que los desarrolladores aumenten la velocidad al configurar un proyecto webpack personalizado. A partir de webpack v4, webpack no espera un archivo de configuración, pero a menudo los desarrolladores desean crear una configuración de paquete web más personalizada en función de sus casos de uso y necesidades. La CLI de webpack aborda estas necesidades al proporcionar un conjunto de herramientas para mejorar la configuración de la configuración personalizada de webpack.:

[**@babel/core**](https://www.npmjs.com/package/@babel/core)**:** es una cadena de herramientas que se utiliza principalmente para convertir el código ECMAScript 2015+ en una versión de JavaScript compatible con versiones anteriores en navegadores o entornos actuales y anteriores. Estas son las principales cosas que Babel puede hacer por usted:

[**@babel/cli**](https://www.npmjs.com/package/@babel/cli)**:** Babel viene con una CLI incorporada que se puede usar para compilar archivos desde la línea de comandos.

Además, varios scripts de puntos de entrada viven en el paquete de nivel superior en @babel/cli/bin. Hay un script de utilidad ejecutable por shell, babel-external-helpers.js, y el script principal de Babel cli, babel.js.

[**@babel/preset-env**](https://www.npmjs.com/package/@babel/preset-env)**:** @babel/preset-env es un preajuste inteligente que le permite utilizar el último JavaScript sin necesidad de microgestión de qué transformaciones de sintaxis (y opcionalmente, polyfills del navegador) son necesarios para su (s) entorno (s) objetivo. ¡Esto te hace la vida más fácil y los paquetes de JavaScript más pequeños!

[**@babel/polyfill**](https://www.npmjs.com/package/babel-polyfill): Babel incluye un polyfill que incluye un tiempo de ejecución de regenerador personalizado y core-js.

Esto emulará un entorno ES2015 + completo y está destinado a ser utilizado en una aplicación en lugar de una biblioteca / herramienta. Este polyfill se carga automáticamente cuando se usa babel-node.

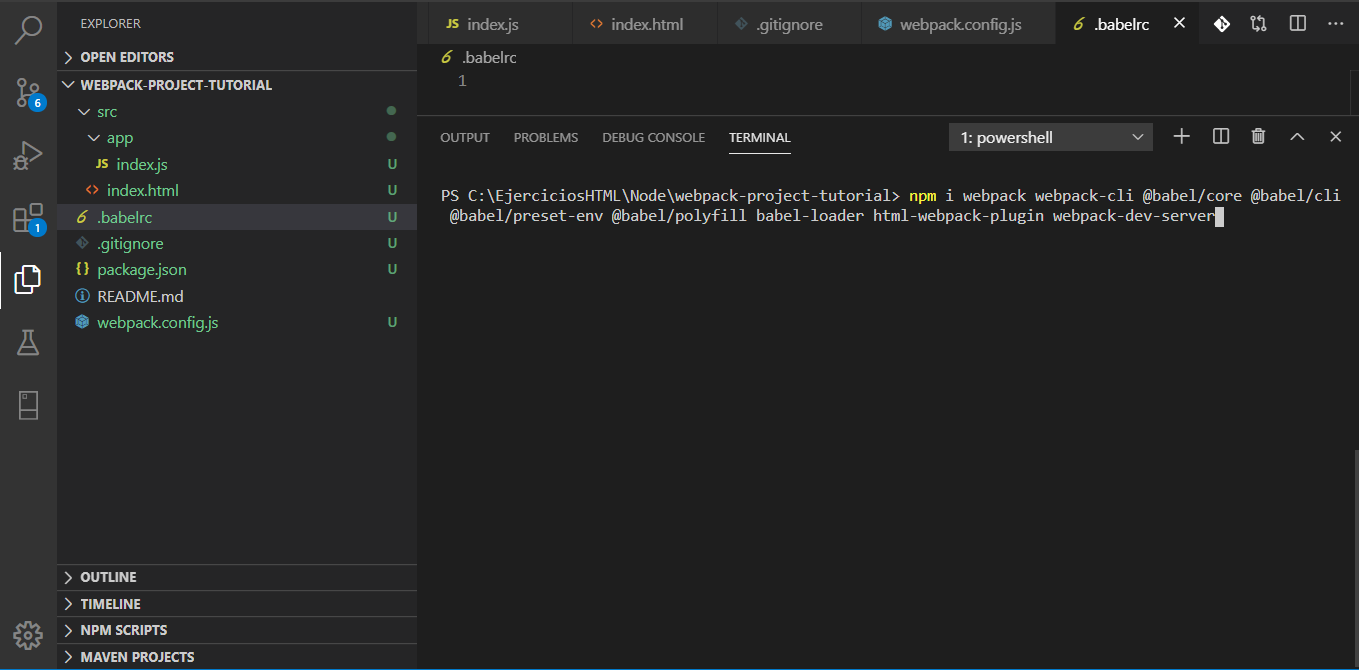
[**babel-loader**](https://www.npmjs.com/package/babel-loader): Este paquete permite transpilar archivos JavaScript utilizando Babel y webpack.

Nota: Los problemas con la salida deben informarse en el rastreador de problemas de Babel.

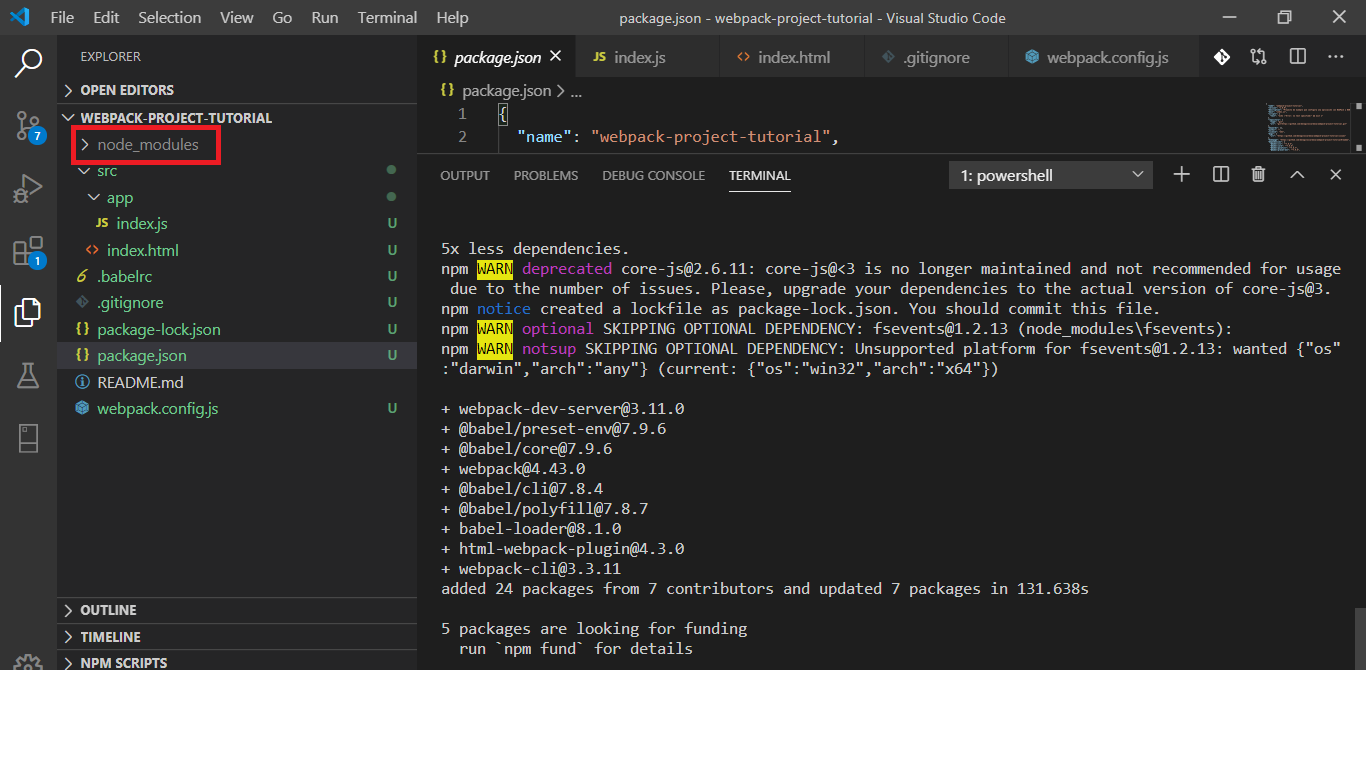
[**html-webpack-plugin**](https://www.npmjs.com/package/html-webpack-plugin): Complemento que simplifica la creación de archivos HTML para servir sus paquetes.

[**webpack-dev-server**](https://www.npmjs.com/package/webpack-dev-server): Use webpack con un servidor de desarrollo que proporcione recarga en vivo. Esto debe usarse solo para el desarrollo.

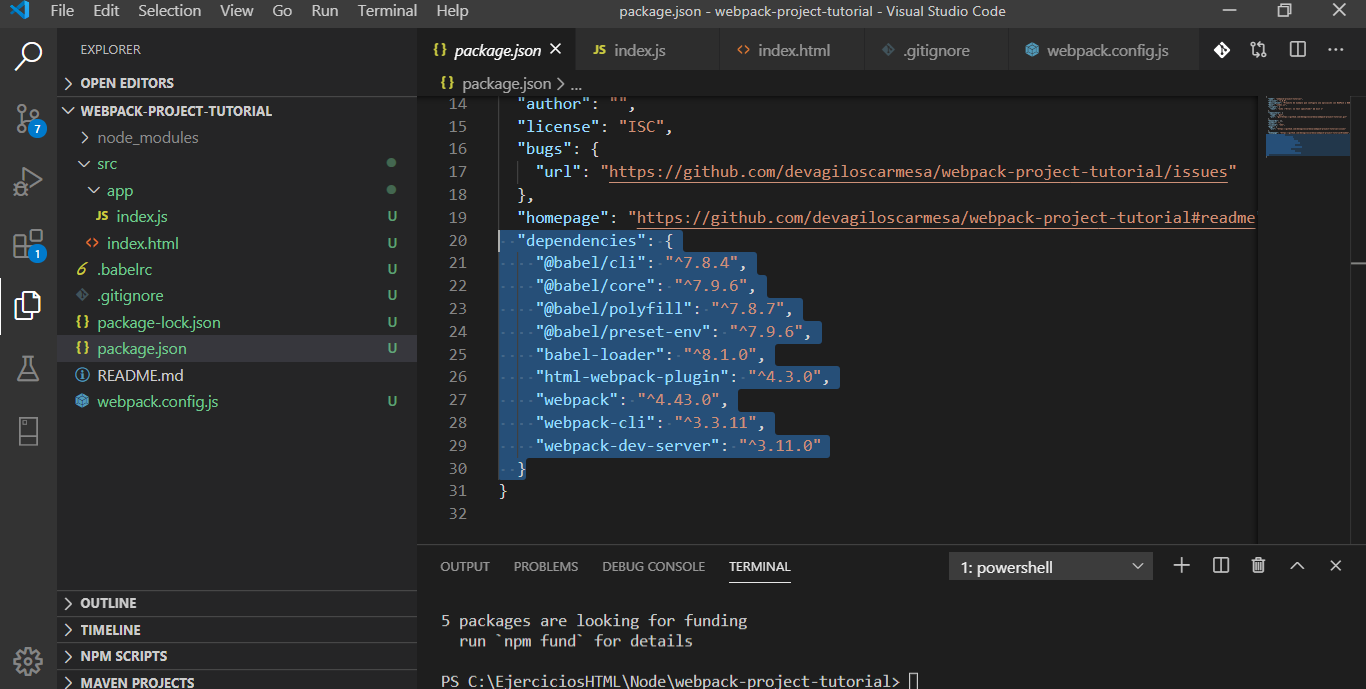
Utiliza webpack-dev-middleware debajo del capó, que proporciona acceso rápido en memoria a los activos del paquete web.



1. Una vez se complete la instalación de estos módulos, se crea una carpeta /node\_modules, en la cual se descargan y almacenan estos módulos para ser utilizados en el proyecto.

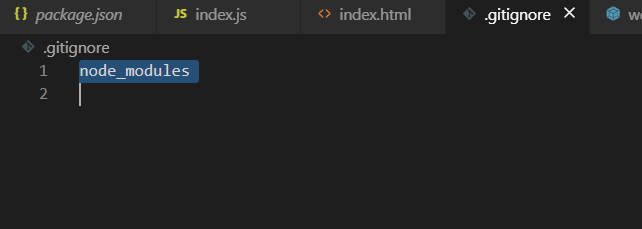


1. También se puede visualizar el archivo /package.json done se adicionan las dependencias con las versión más estable que se pueda instalar en la maquina donde se corrió la instalación.



1. Debido a que estas dependencias quedan configuradas en el archivo /package.json, la carpeta de /node\_modules no puede ser sincronizada en el repositorio, debido a su peso y que en cada maquina puede presentar diferentes instalaciones, ya sea por su sistema operativo (Windows, Linux y todas sus versiones, Mac OS, etc), entre otras factores. Por ello se debe adicionar este directorio en el archivo /.gitignore .

/.gitignore: En el gitignore se especificarán todas las rutas y archivos que no se requieren y con ello, el proceso de control de versiones simplemente ignorará esos archivos. Es algo tan habitual que no debíamos de dejarlo pasar en el Manual de Git.



1. Procedemos a realizar las configuraciones pertinentes de webpack, para ello adicionaremos el siguiente código JS en el archivo / webpack.config.js.

const path = require('path');

module.exports = {

entry: './src/app/index.js',

output: {

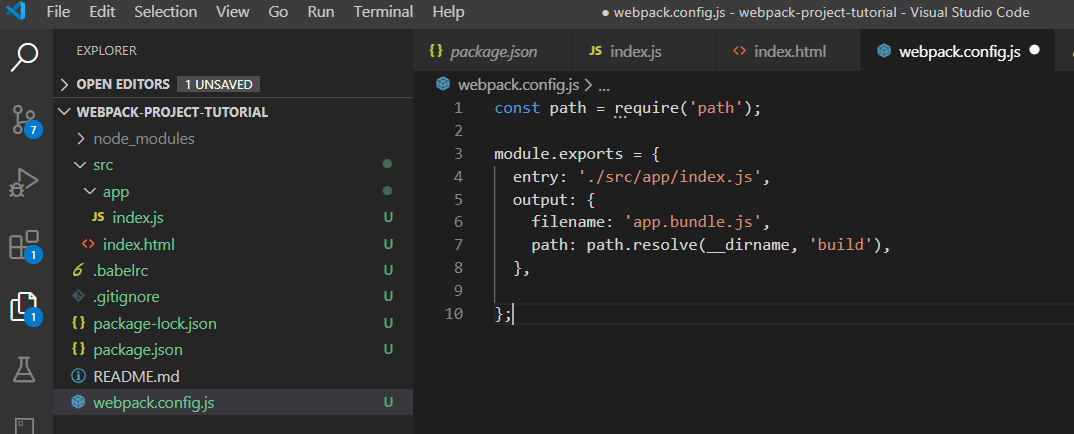
filename: 'app.bundle.js',

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

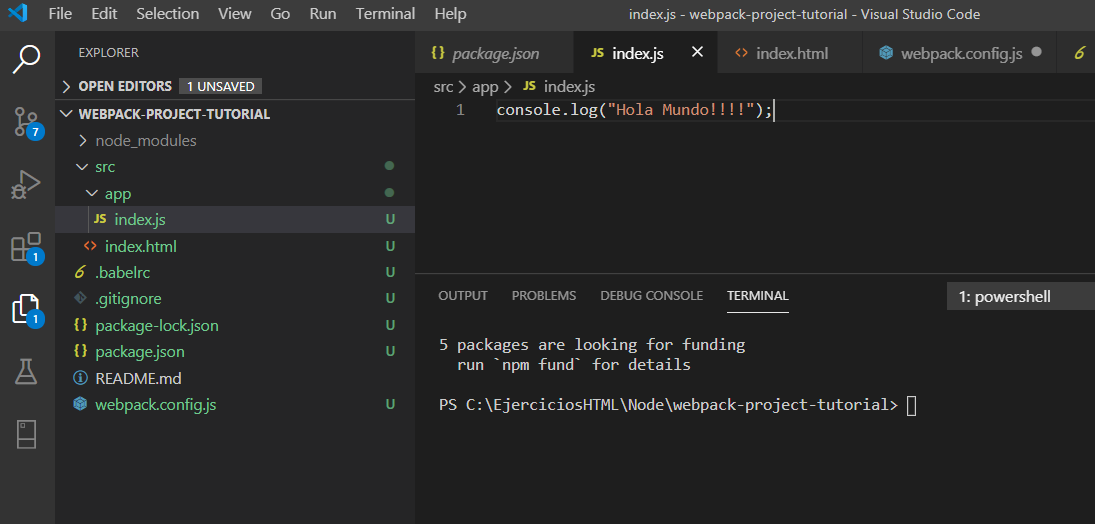
},

};

En este archivo se deben adicionar las reglas y demás configuraciones de webpack, por ahora solo se compilara el archivo /src/app/index.js en un archivo app.bundle.js en la carpeta build.

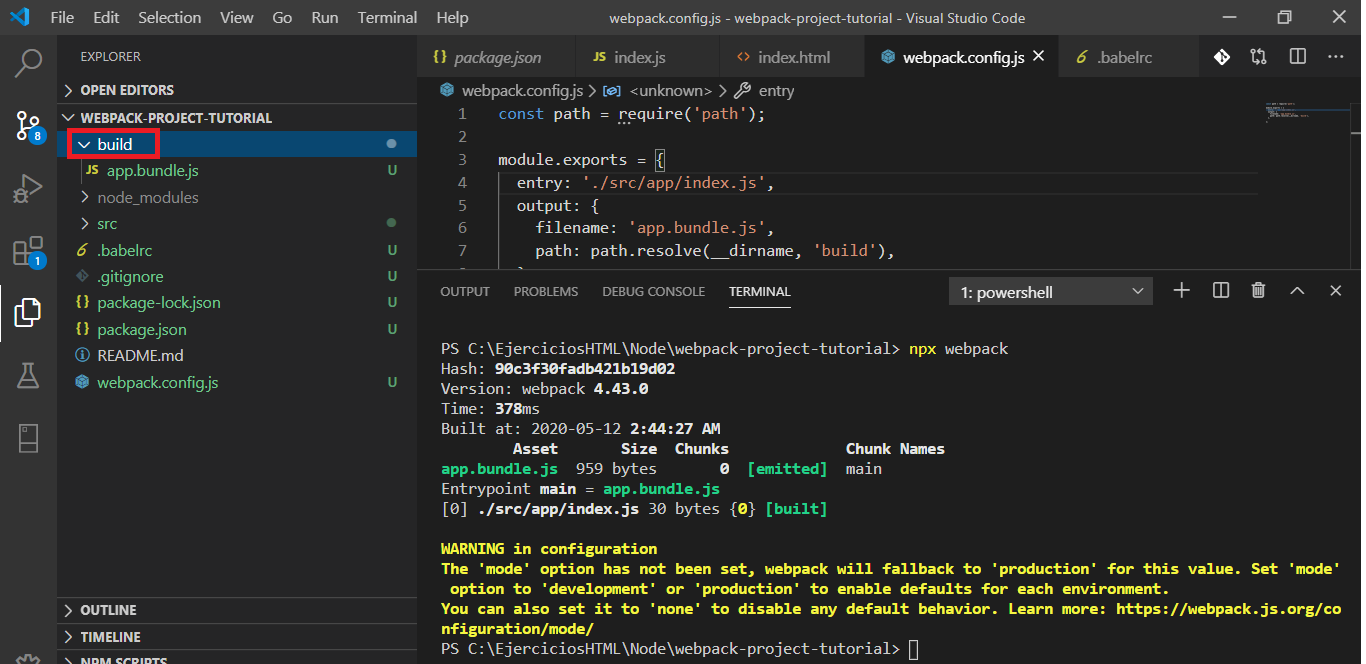


1. Una vez se almacena el archivo /webpack.config.js se puede compilar el proyecto, para efectos de prueba adicionemos algún ejemplo en el archivo /src/app/index.js.



Luego procedemos a ejecutar el siguiente comando para compilar el proyecto.

npx webpack



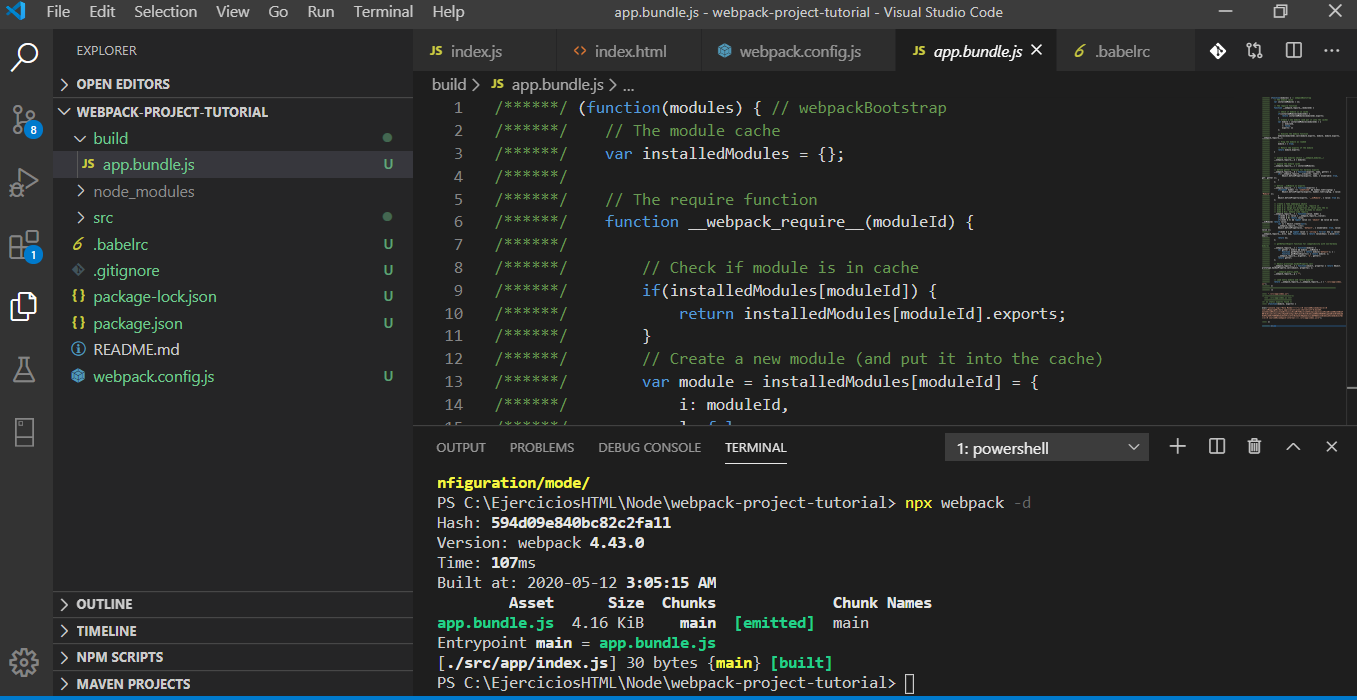
Una vez se compila el proyecto, se puede visualizar el directorio /build. Allí se encuentra de momento únicamente /build/app.bundle.js , archivo donde se munífica y comprime el contenido de los archivos JS configurados en el archivo /webpack.config.js.

1. El mensaje que aparece, indica que el comando npx webpack, se puede ejecutar en diferentes ambientes, con lo cual se logra que la compresión del proyecto cambie, por defecto la compresión o compilado del proyecto se hace en modo producción, pero también se puede hacer en modo desarrollo; los siguientes comandos son los que se poden emplear para cambiar la compilación:

module.exports = {

mode: 'development',

};



En este caso se puede ver que el archivo /build/app.bundle.js cambia la apariencia, debido a que se está ejecutando en modo desarrollo.

1. A continuación, probaremos algunas configuraciones que se pueden realizar con webpack, para ello se configurará el plugin html-webpack-plugin, con este se simplificara la creación del archivo /src/index.html, una vez se compile el proyecto el archivo queda en la ruta /build/index.html, tal como se especifica en el archivo de configuración de webpack. Se puede profundizar en la página de webpack, donde explican la configuración de un [plugin](https://webpack.js.org/plugins/).

const path = require('path');

const HTMLWebpackPlugin = require('html-webpack-plugin')

module.exports = {

entry: './src/app/index.js',

mode: 'development',

output: {

filename: 'app.bundle.js',

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

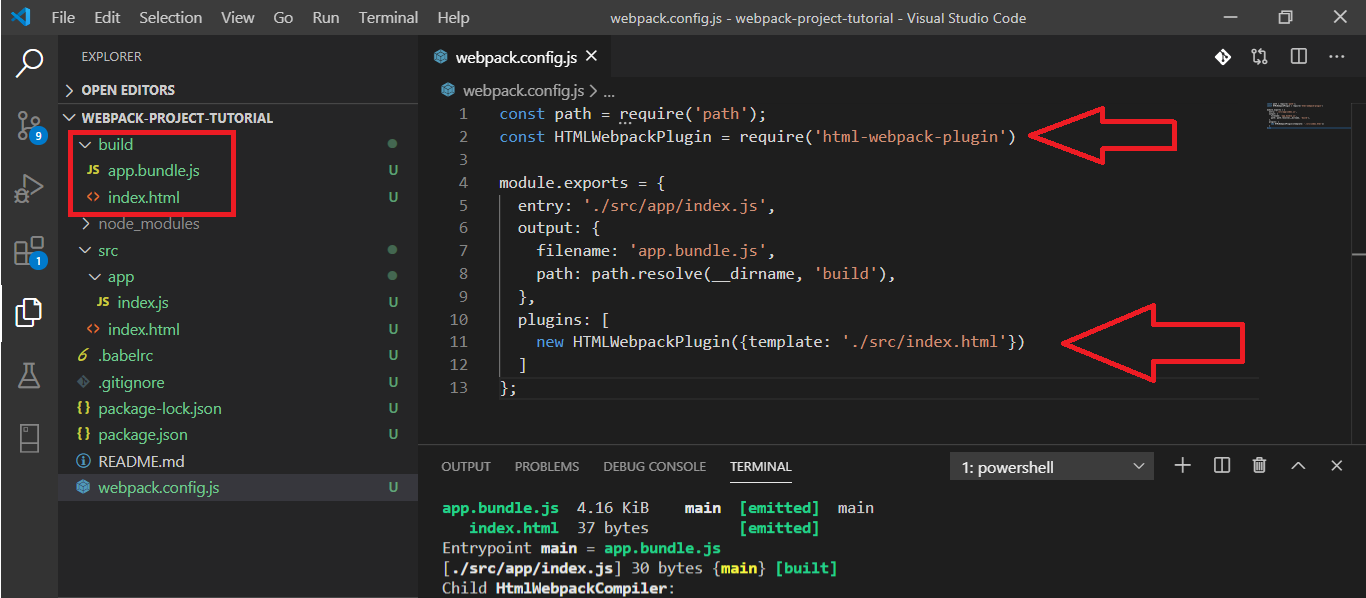
},

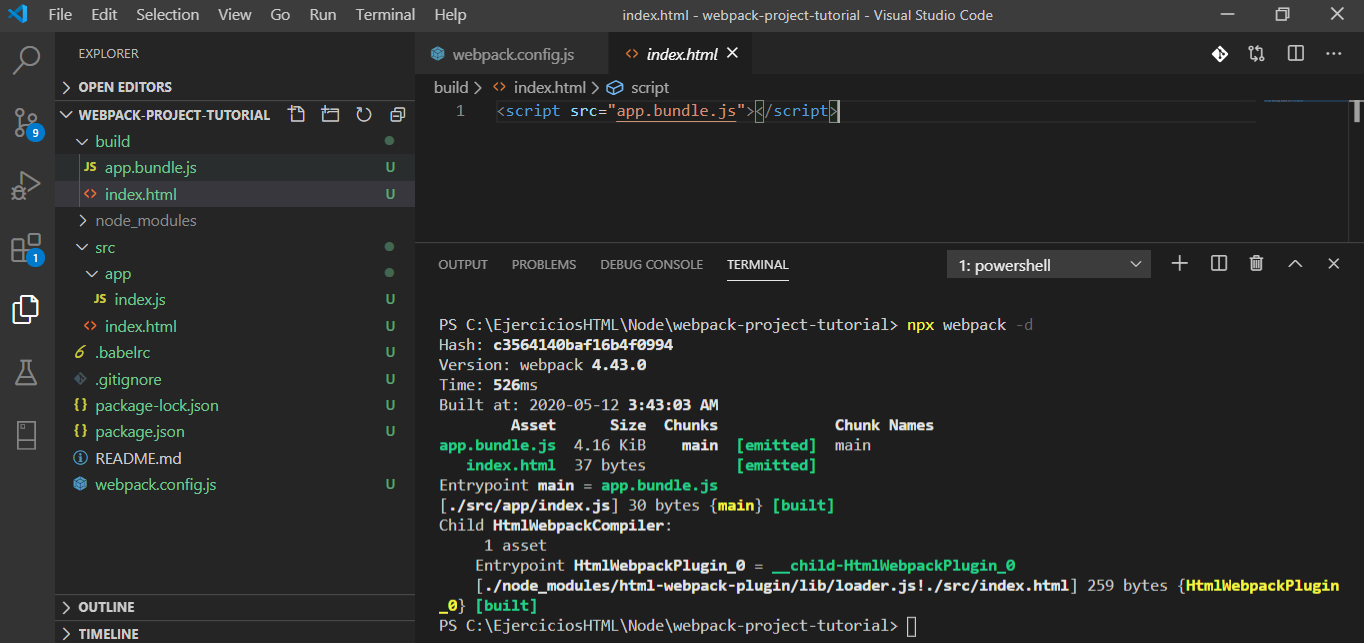
plugins: [

new HTMLWebpackPlugin({template: './src/index.html'})

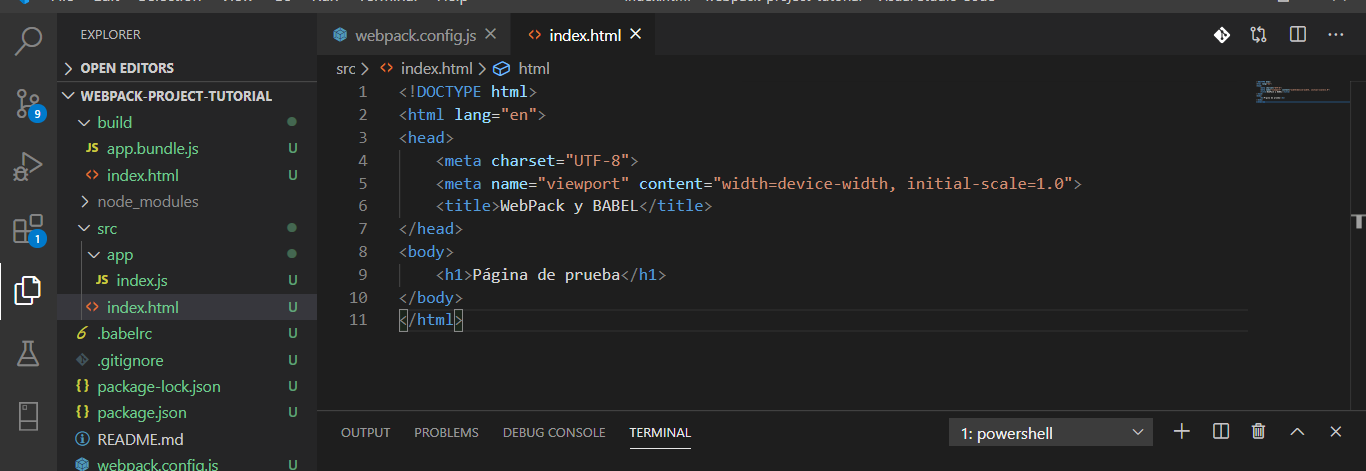
]

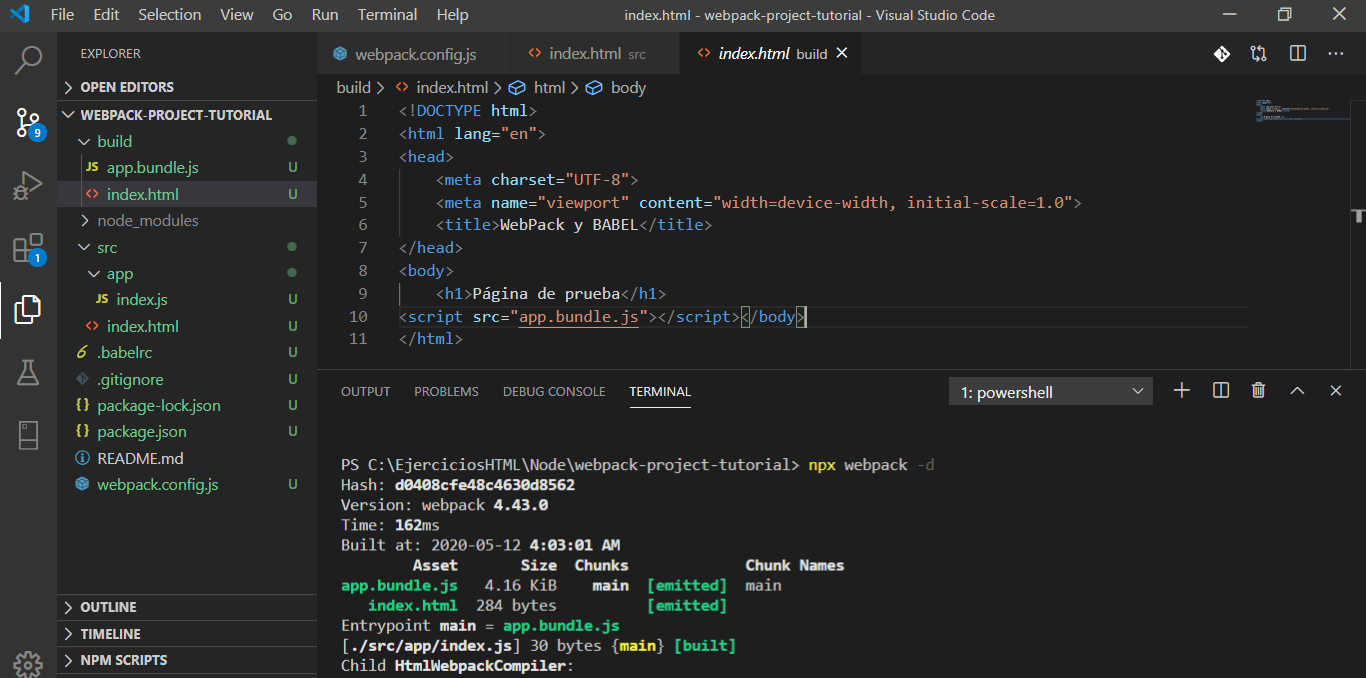
};





1. Una vez se adiciona algún ejemplo en la página /src/index.html, la salida adiciona automáticamente el archivo JS, como se evidencia a continuación.





1. También se puede comprimir los archivos html, para los cual se debe cambiar una configuración en el archivo de configuración de webpack. Dicha configuración se corresponde al plugin html-webpack-plugin, la cual emplea mitificación sobre los archivos que se hayan configurado, esto se puede ver en el apartado del repositorio, donde se encuentra la [documentación](https://github.com/jantimon/html-webpack-plugin) de este complemento. Para ello se adiciona el siguiente JSON como parámetro al momento de instanciar un objeto de este plugin.

minify: {

collapseWhitespace: true,

removeComments: true,

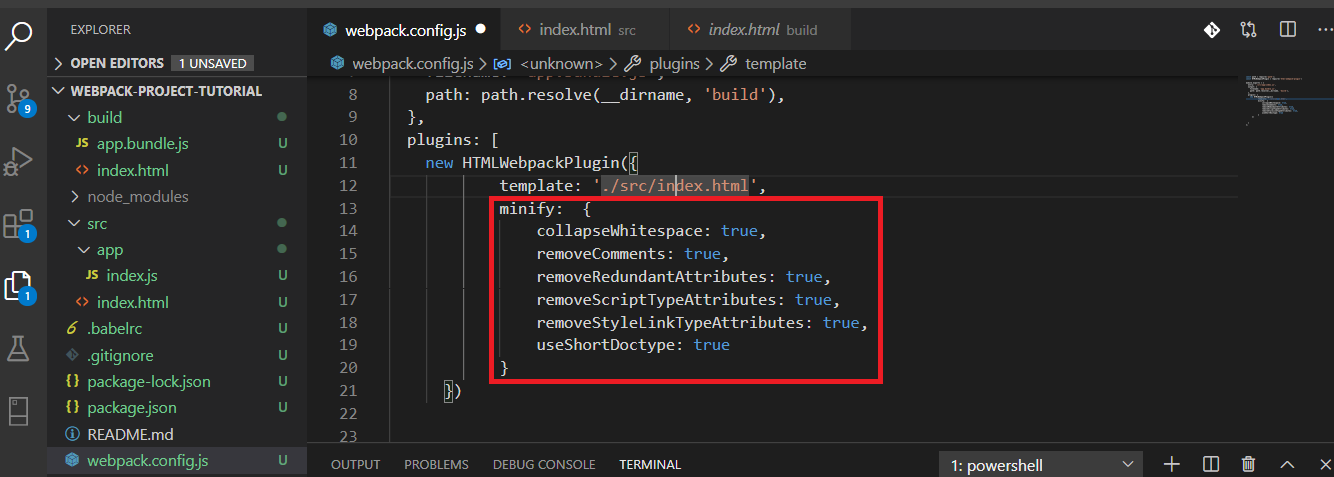
removeRedundantAttributes: true,

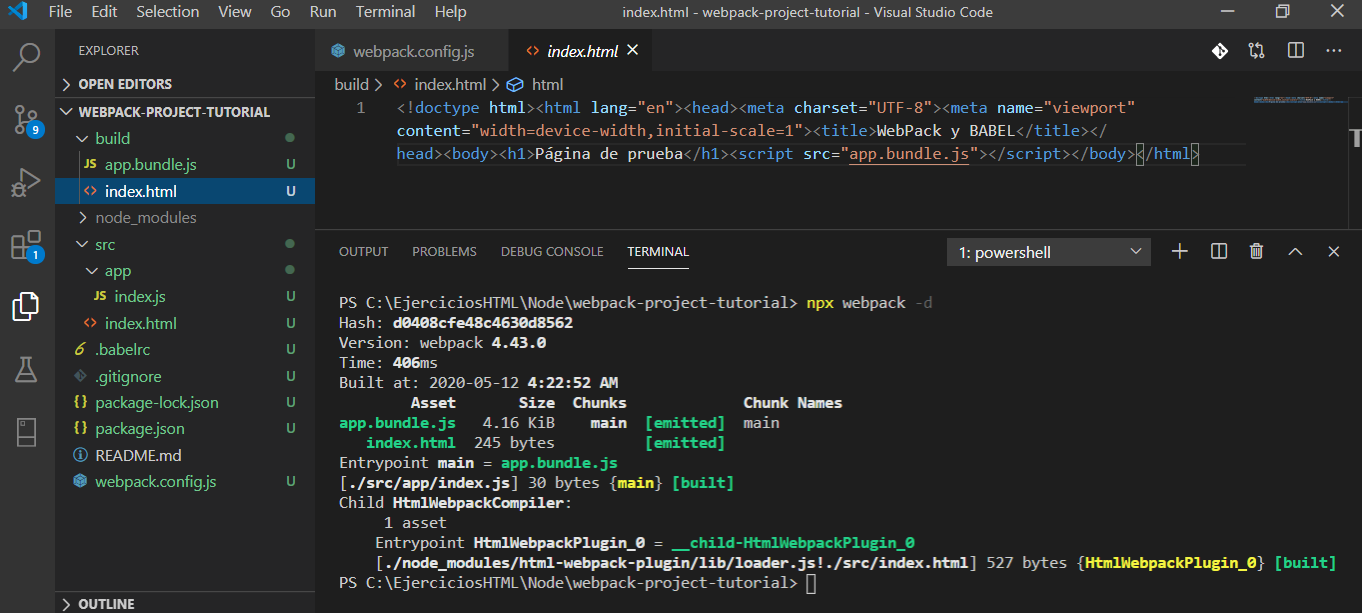
removeScriptTypeAttributes: true,

removeStyleLinkTypeAttributes: true,

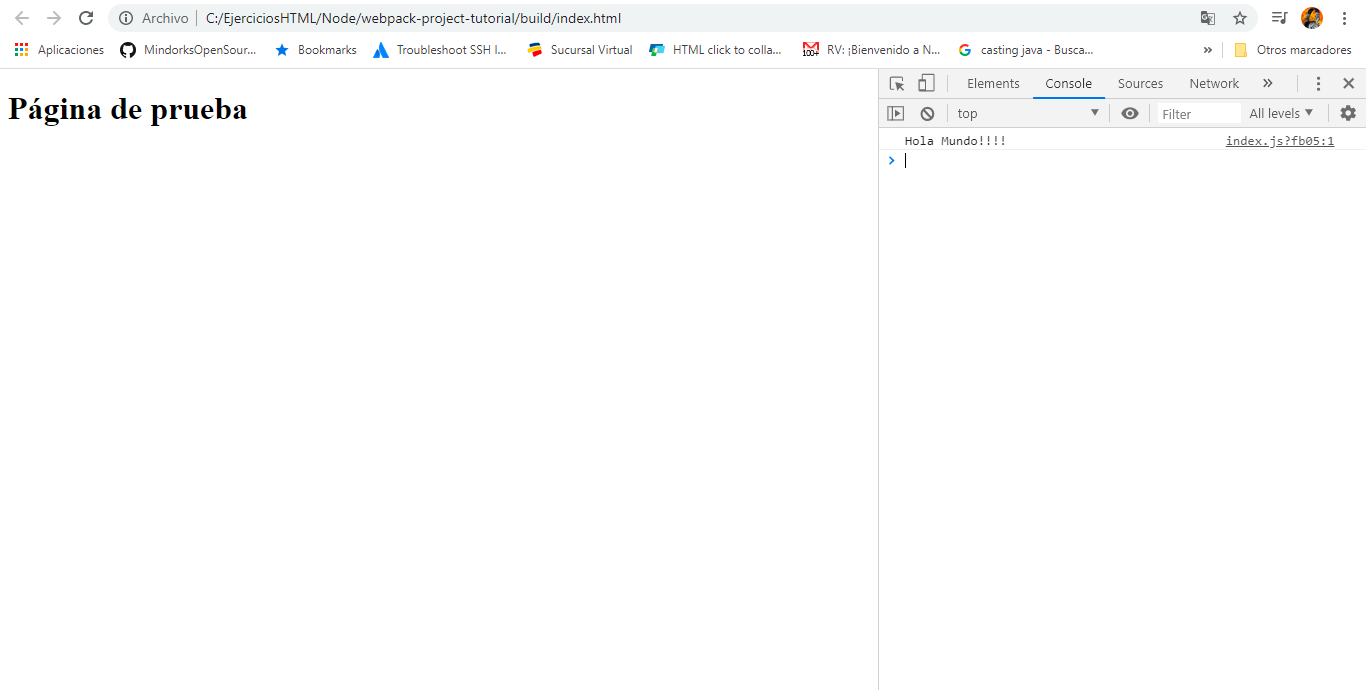
useShortDoctype: true

}





1. Se puede abrir la página directamente desde el proyecto y se observa que no existe ningún error.



1. A continuación, veremos algunas ventajas que tiene WebPack, respecto a la sintaxis moderna de JS. Para ello definiremos una clase llamada Comentario, definida en la ruta /src/app/comentario.js, de igual forma se puede exportar la clase e importarlo desde otros archivos. Los siguientes ejemplos se definen en cada uno de los archivos que estamos creando.

**/src/app/comentario.js**

class Comentario{

obtenerComentarios(){

return [];

}

}

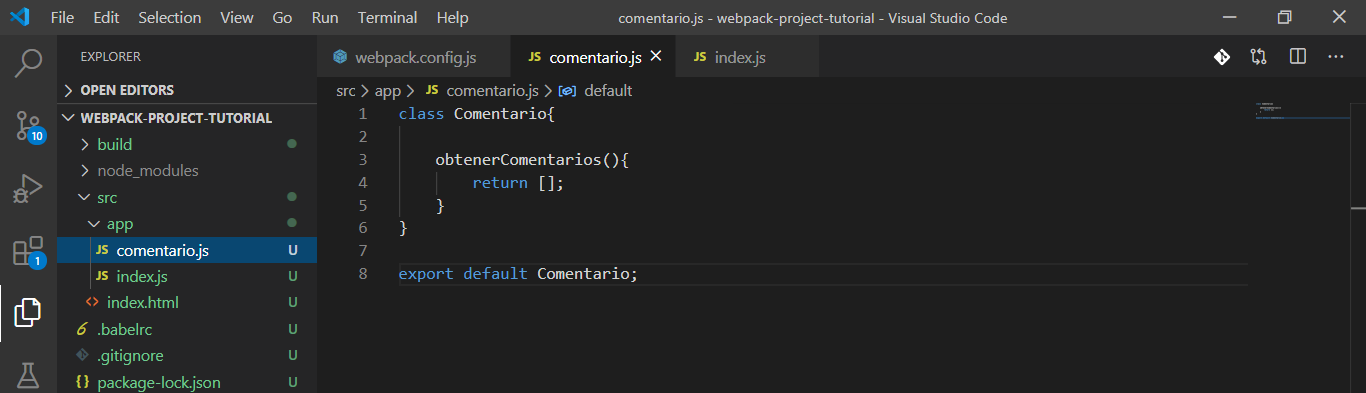
export default Comentario;

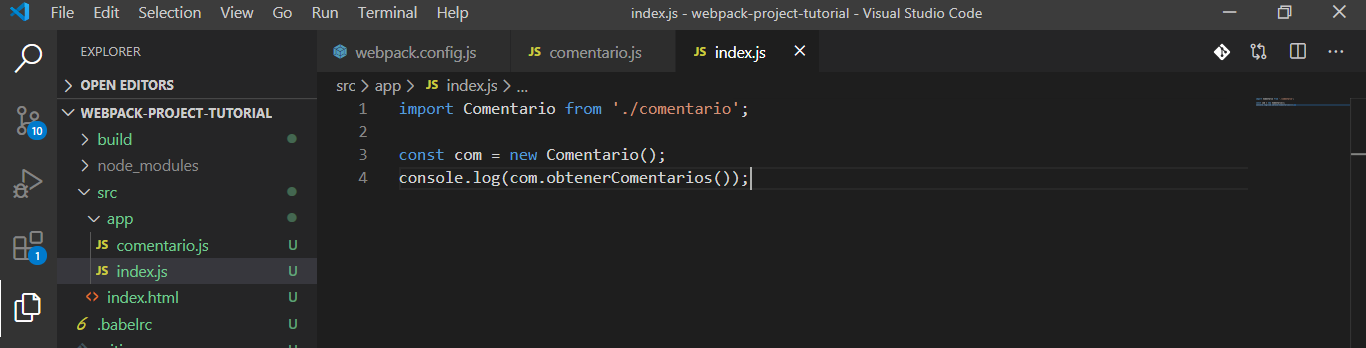
**/src/app/index.js**

import Comentario from './comentario';

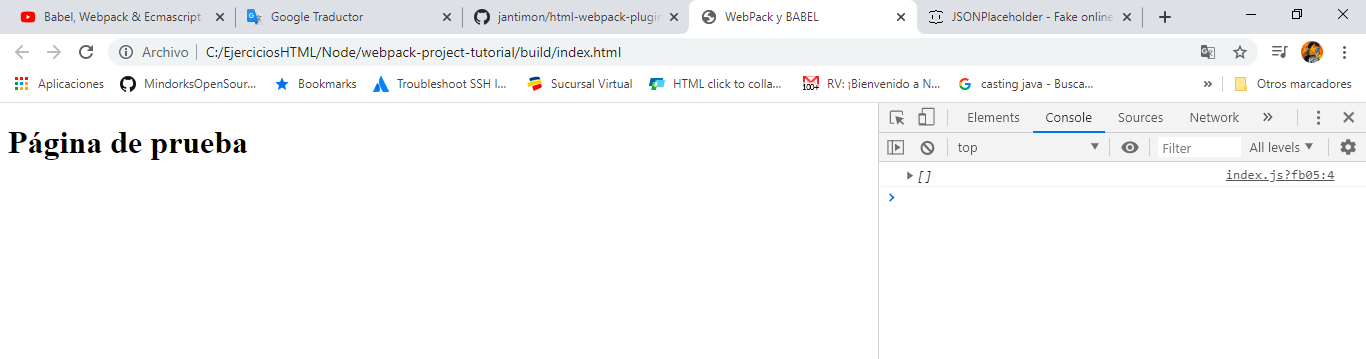
const com = new Comentario();

console.log(com.obtenerComentarios());

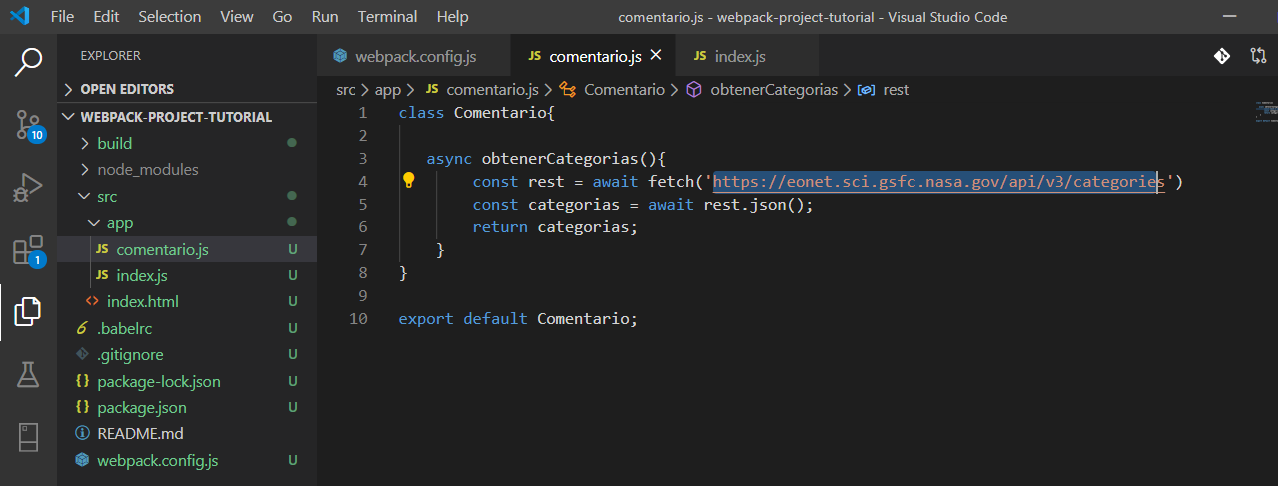


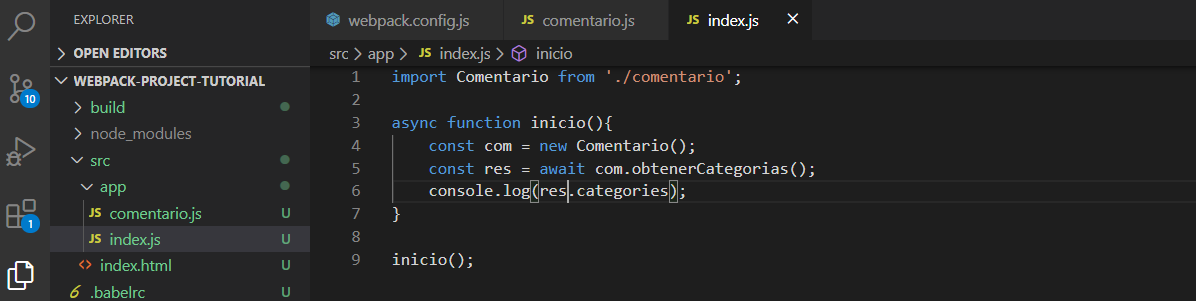


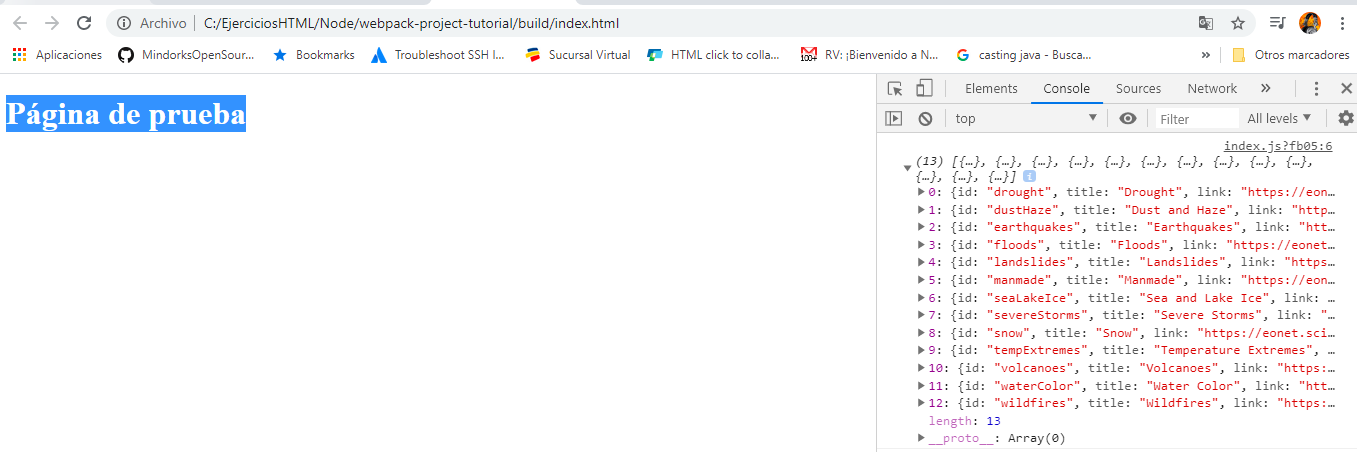
Una vez se compile nuevamente el proyecto, con el comando npx webpack -d podemos visualizar la salida en el navegador y notaremos que estamos usando la sintaxis nueva de JS y no tendremos problemas como lo vimos en los inicios del curso.



1. A continuación, veamos algunas mejoras que podemos hacer al código, consumiendo un servicio que se encuentra en un sitio [nas a](https://eonet.sci.gsfc.nasa.gov/how-to-guide), sitio que permite consumir datos de prueba para ensayar nuestras aplicaciones. En este caso consumiremos el api de categorías con la siguiente URL: <https://eonet.sci.gsfc.nasa.gov/api/v3/categories>. Para ello se empleara el async/await y la función fetch para obtener los datos.

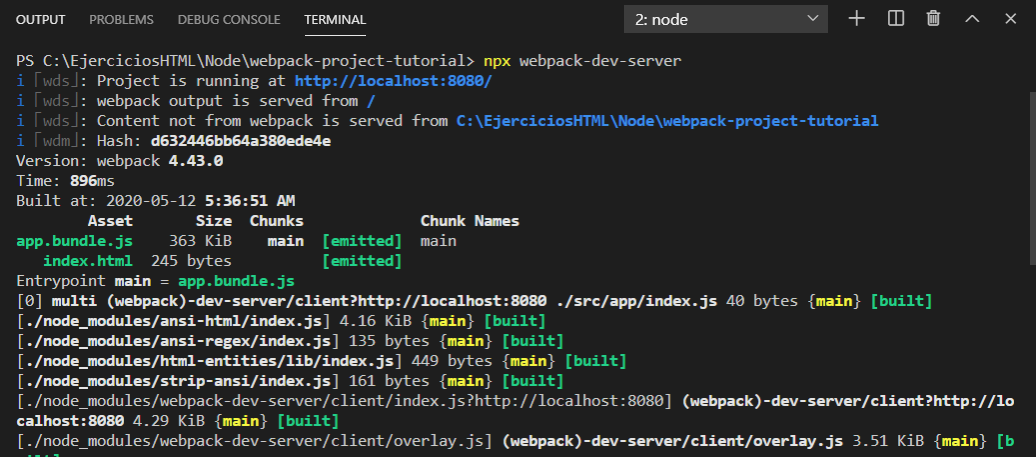




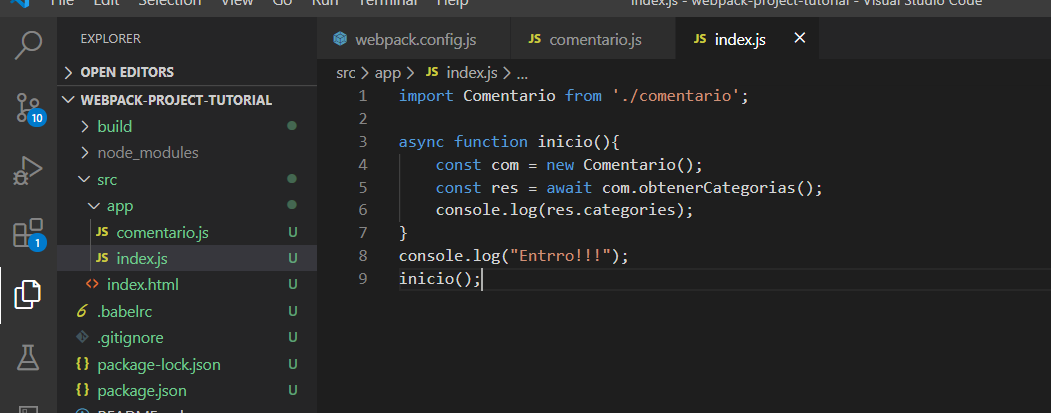


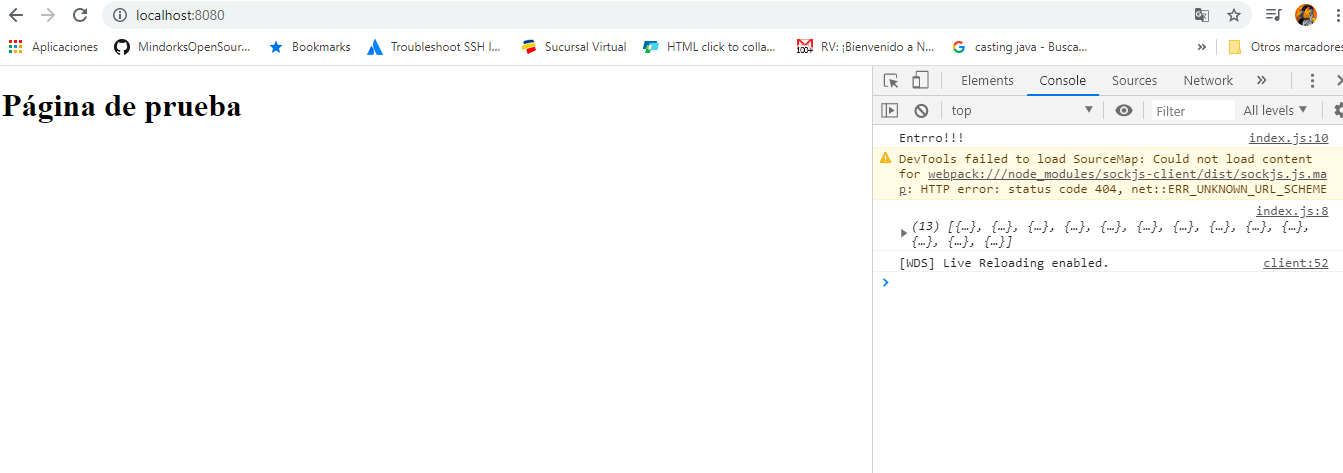
1. A continuación, configuraremos un servidor de desarrollo con otro de los complementos que instalamos previamente, esta dependencia es webpack-dev-server con esto lograremos tener un servidor que este escuchando los cambios que se realizan en el proyecto webpack. Para ello se puede ejecutar el siguiente comando:

npx webpack serve

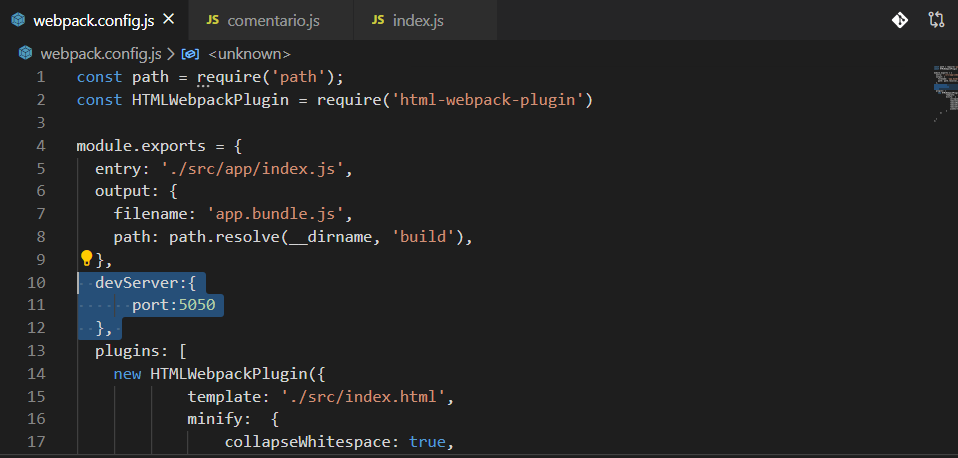


De esta manera se ejecuta un servidor que está escuchando en el puerto 8080, para ver la salida entramos a la siguiente ruta <http://localhost:8080>. Con esta dependencia se logra que cualquier cambio que hagamos en el proyecto, se visualice inmediatamente, es decir, que no tenemos que recompilar manualmente.

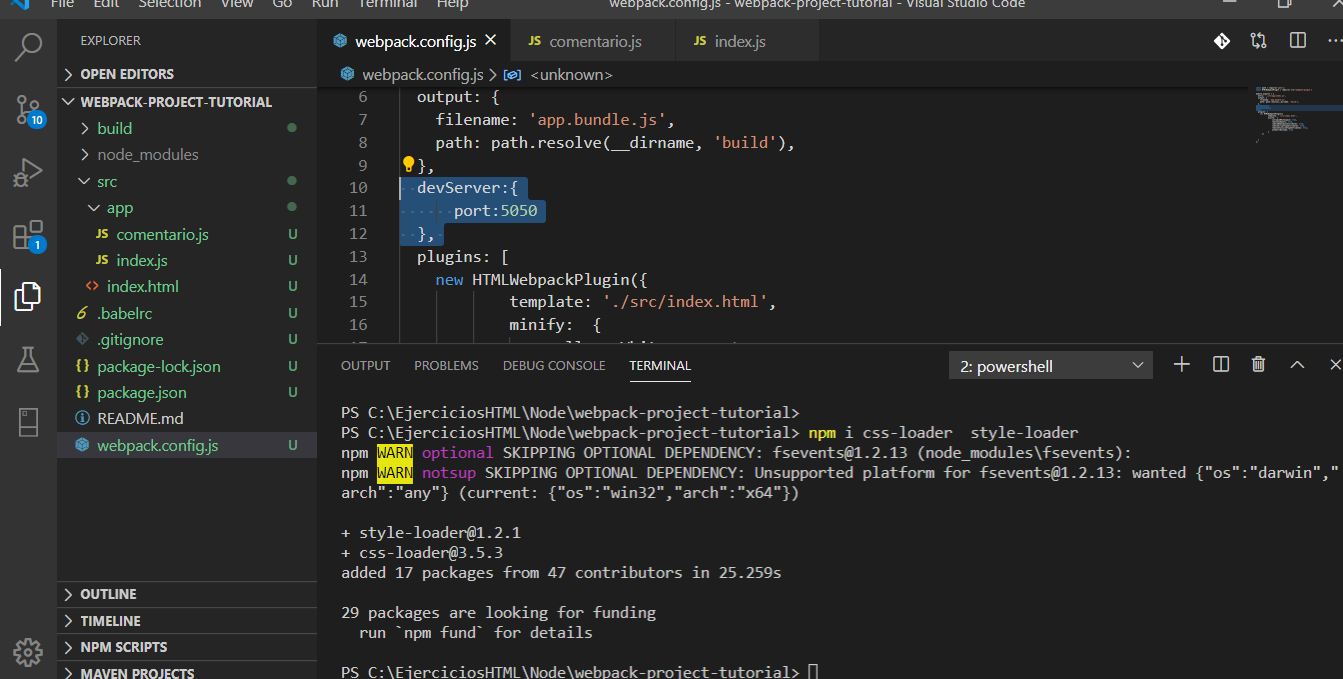




De igual forma, se puede realizar algunas configuraciones en el archivo webpack.config.js, que apliquen para la configuración del servidor, para ello realizamos la siguiente configuración.



1. A continuación, usaremos otro plugin, el cual servirá para importar archivos css dentro de los archivos JS que se compilen, dicho plugin es y style-loader, para su instalación ejecutaremos en consola el siguiente comando para instalar dicha dependencia: yarn add css-loader style-loader



1. Luego se deben agregar las siguientes reglas al archivo webpack.config.js, con lo cual, se indica con una expresión regular, que teste los archivos que terminen en .css y en dichos casos use las dos dependencias instaladas anteriormente. El siguiente código JSON, es el que se debe agregar en el archivo.

module:{

rules: [

{

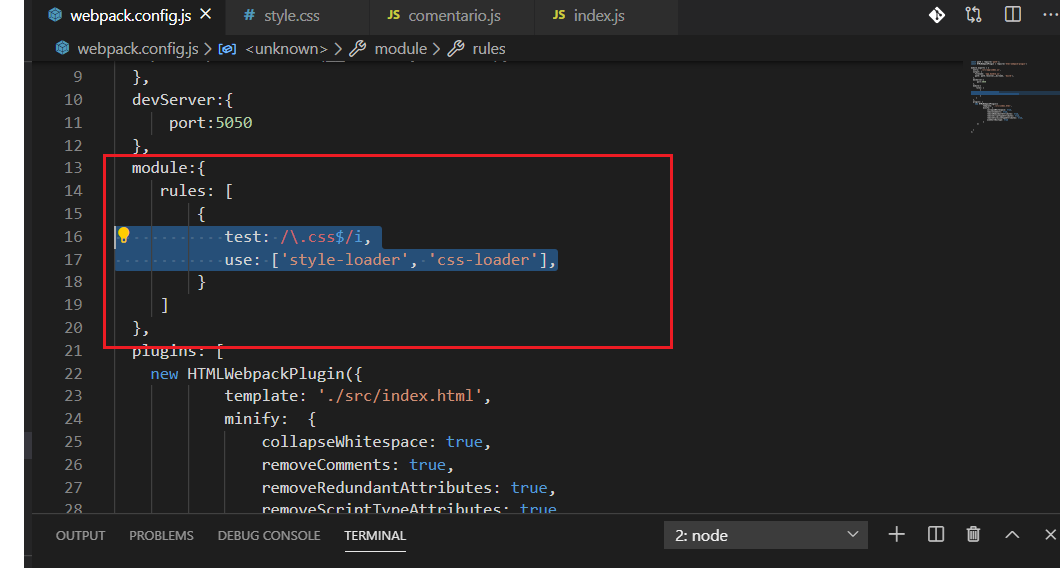
test: /\.css$/i,

use: ['style-loader', 'css-loader'],

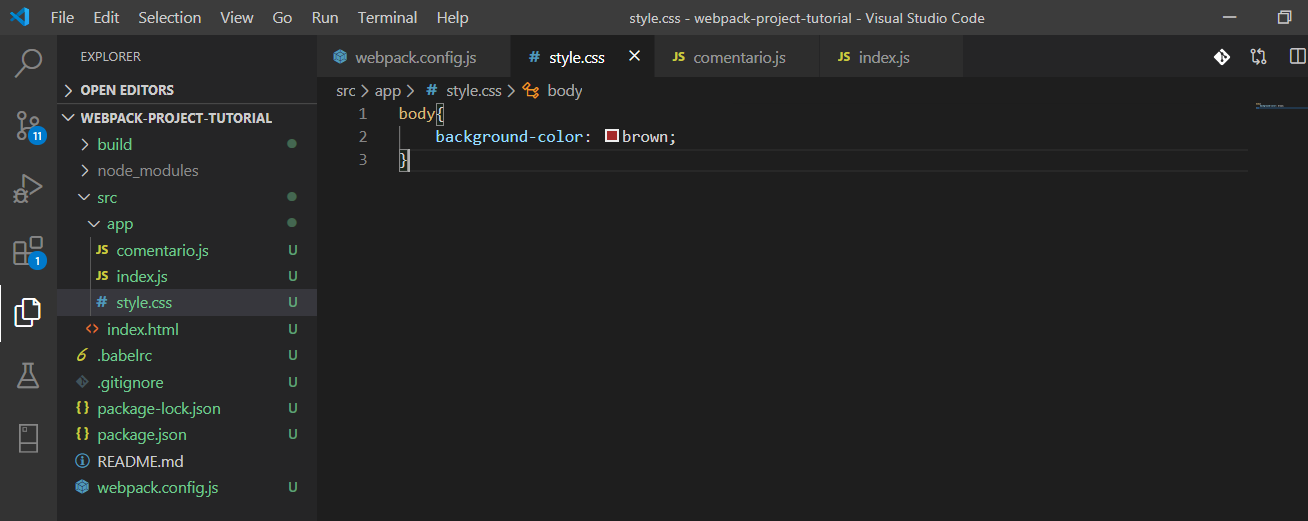
}

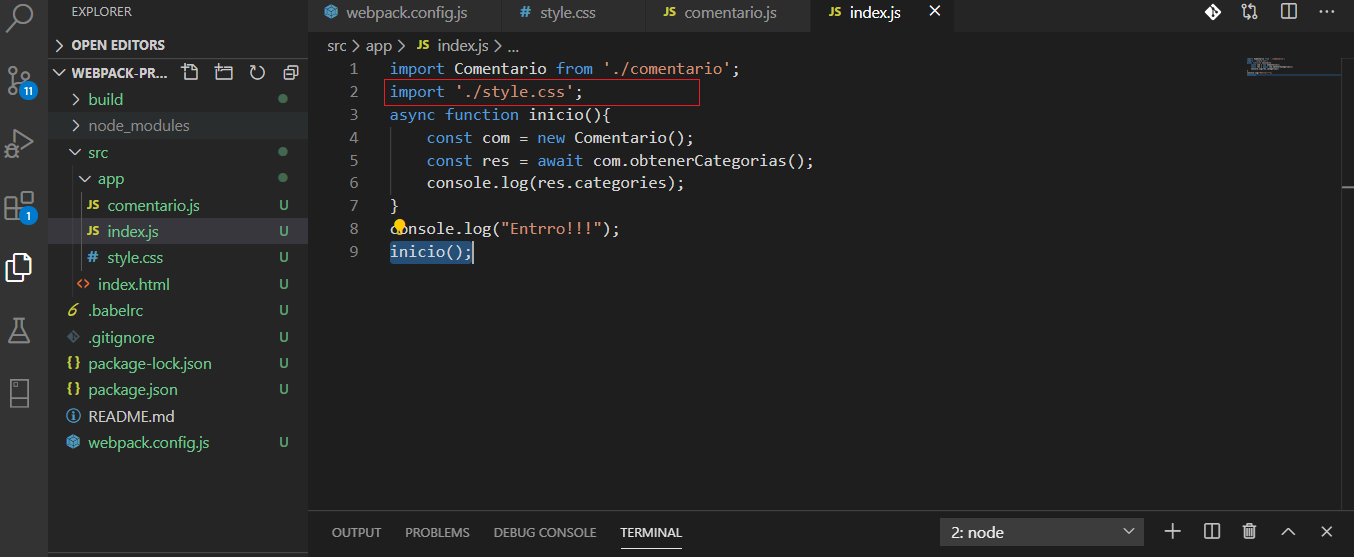
]

}



1. A continuación, creamos el siguiente archivo css /src/app/style.css, el cual lo importaremos desde nuestro archivo JS.

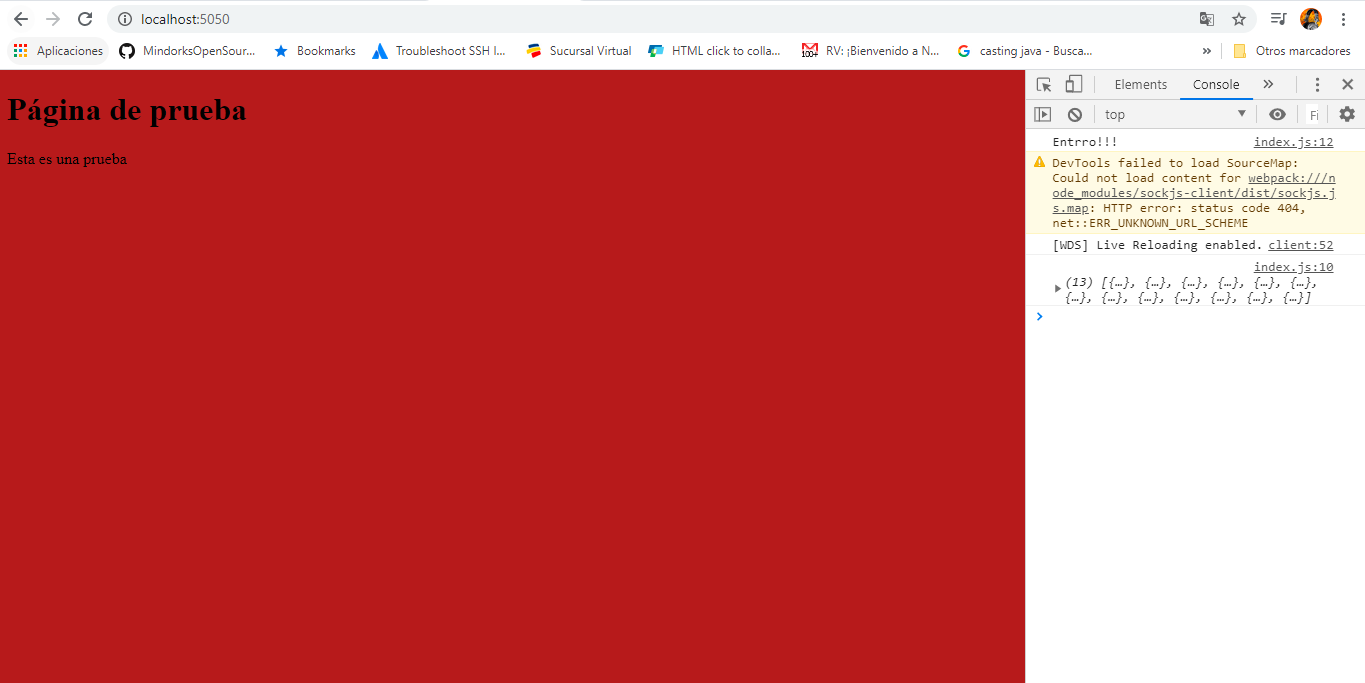




1. Volvemos a ejecutar los comandos para compilar y desplegar el servidor:

npx webpack -d

npx webpack-dev-server



1. A continuación, separaremos los archivos JS y CSS, debido a que al compilar el proyecto todo el código CSS y JS queda dentro del archivo /build/ app.bundle.js, de este modo se logra que no todo quede concentrado en un único archivo. Para ello, se debe usar el siguiente plugin para realizar extracción de css y de este modo poder lograr una separación en la compilación del proyecto, dicho plugin es [mini-css-extract-plugin](https://www.npmjs.com/package/mini-css-extract-plugin), por lo cual correremos el siguiente comando para lograr instalar este plugin:

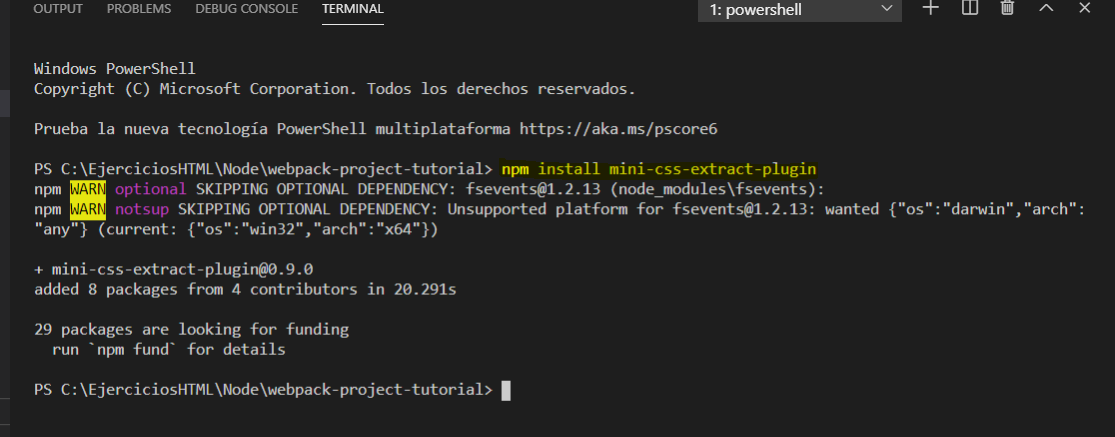
yarn add mini-css-extract-plugin

mini-css- -plugin: Este complemento extrae CSS en archivos separados. Crea un archivo CSS por archivo JS que contiene CSS. Es compatible con la carga a pedido de CSS y SourceMaps.

Se basa en una nueva característica de webpack v4 (tipos de módulos) y requiere webpack 4 para funcionar.

En comparación con el extracto-text-webpack-plugin:

* Carga asíncrona
* No duplicar la compilación (rendimiento)
* Más fácil de usar
* Específico para CSS



1. Luego debemos especificar la configuración en el archivo de configuración de WebPack. Para esto, debemos adicionar el siguiente fragmento de código JS.

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {

plugins: [

new MiniCssExtractPlugin({

filename: 'app.bundle.css'

}),

],

module:{

rules: [

{

test: /\.css$/i,

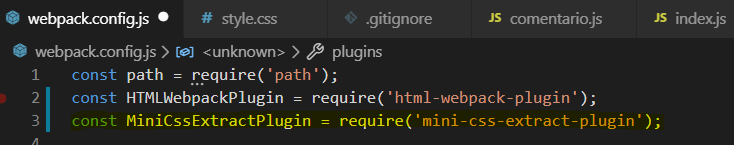
use: [MiniCssExtractPlugin.loader, 'css-loader'],

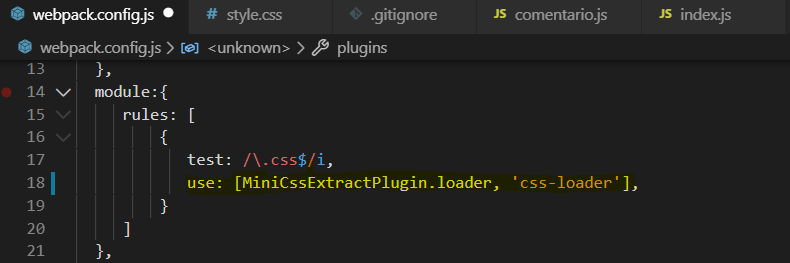
}

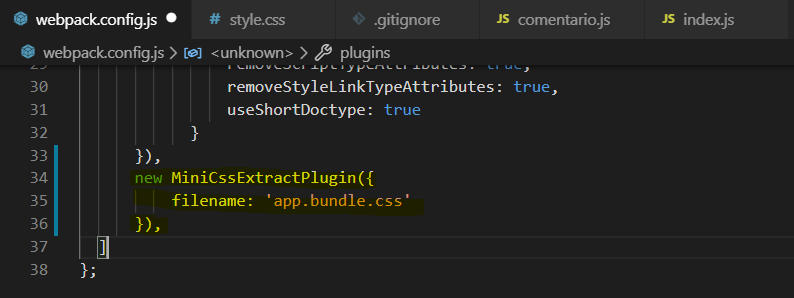
]

}, ,

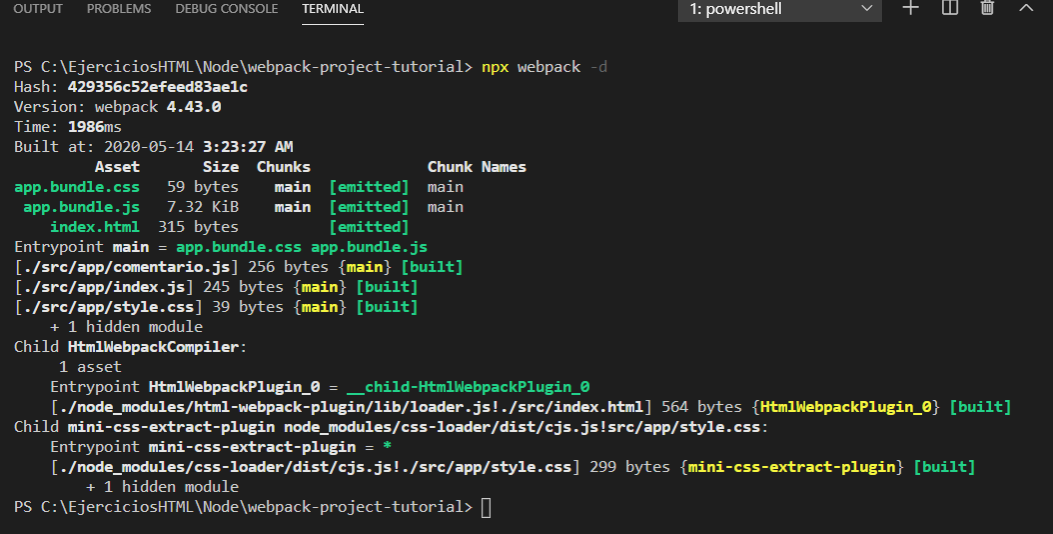
};

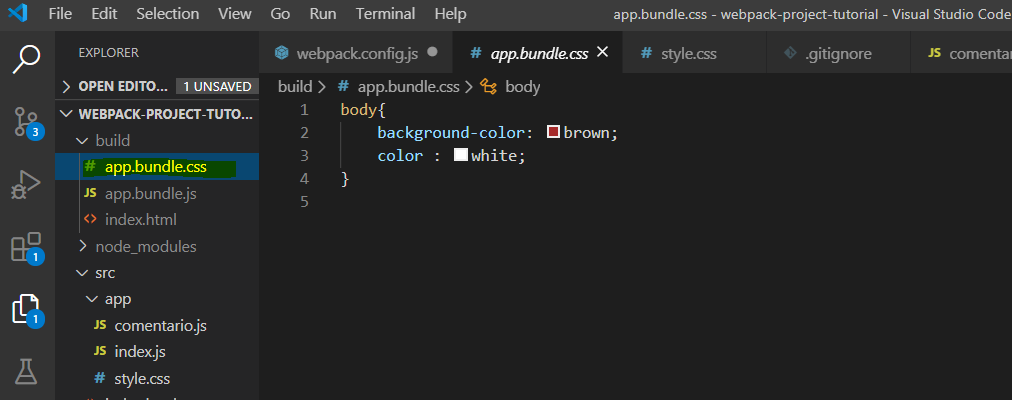






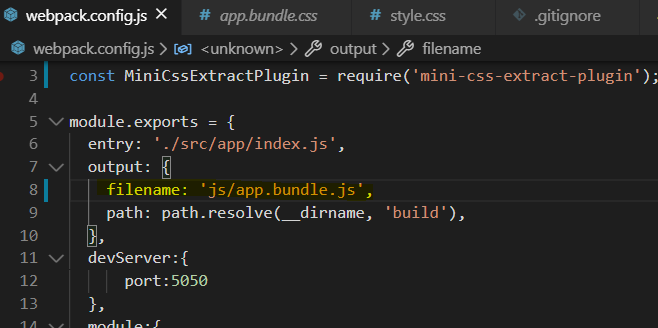
1. Paso seguido volveremos a compilar el proyecto con el fin de evidenciar la extracción del archivo CSS en la carpeta build.

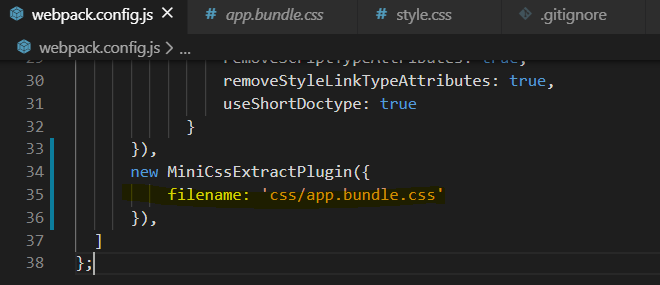




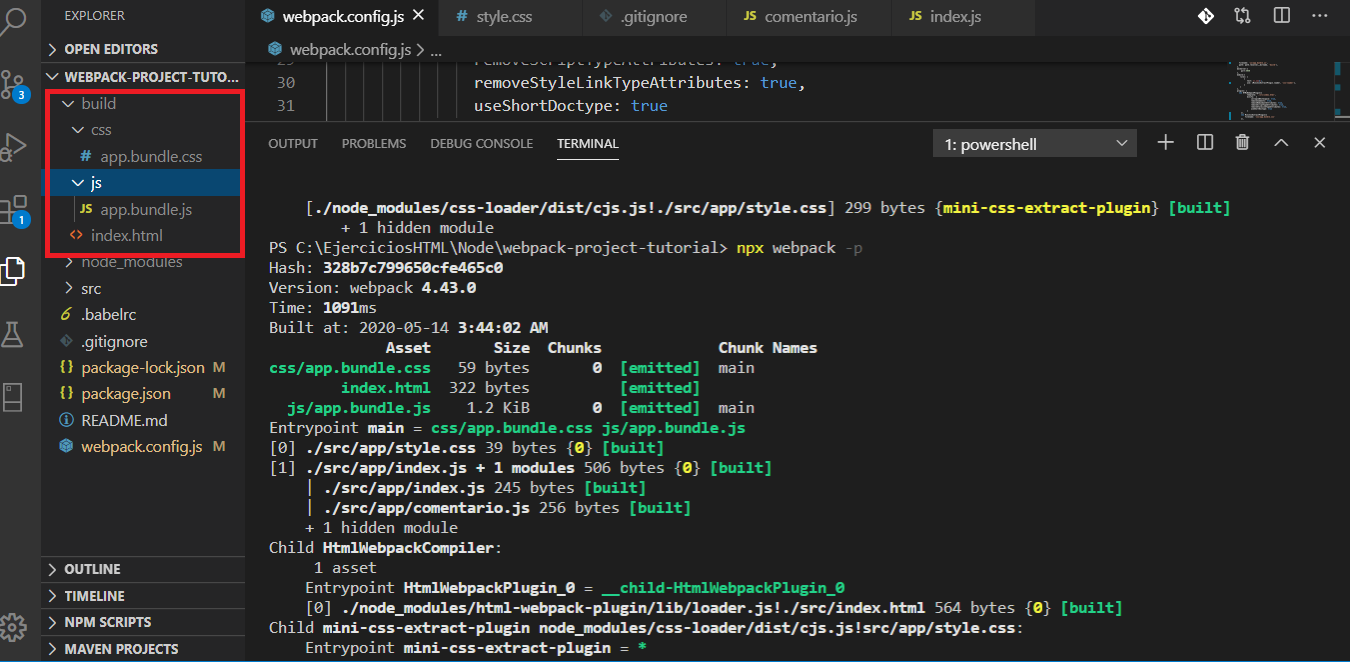
1. Podríamos asignar mayor orden al proyecto de compilación, indicando la ruta de compresión de los archivos css y js, para ello modificamos el archivo de configuración de WebPack, de modo que las rutas queden la siguiente manera:

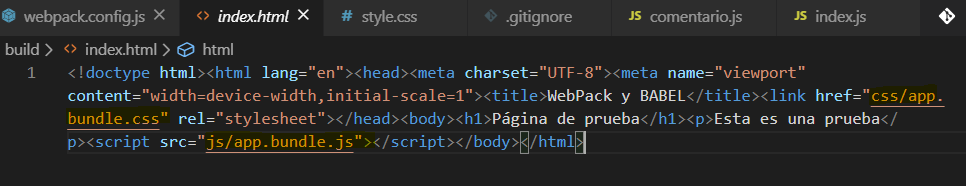
* js/app.bundle.js
* css/app.bundle.css





1. Borre la carpeta de build y volva compilar el proyecto y veremos la nueva estructura en el proyecto.





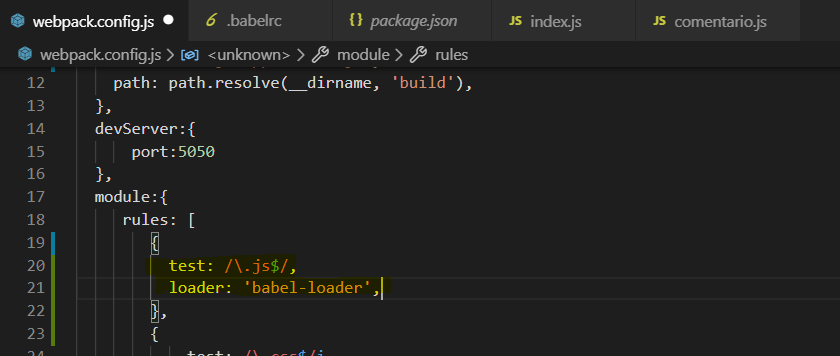
1. A continuación, se configurará BABAL en el proyecto para ver su funcionamiento. Para ello configuraremos el archivo .babelrc (la configuración de este archivo se puede encontrar en su [sitio](https://babeljs.io/docs/en/6.26.3/babelrc)). Iniciaremos configurando el archivo de configuración de webpack, agregando la siguiente regla para que verifique todos los archivos JS y convierta el código que no es reconocible por los diferentes navegadores.

{

test: /\.js$/,

loader: 'babel-loader',

}

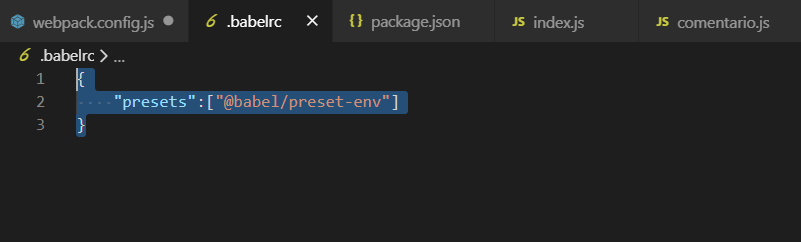


1. Luego adicionamos la siguiente configuración en el archivo .babelrc

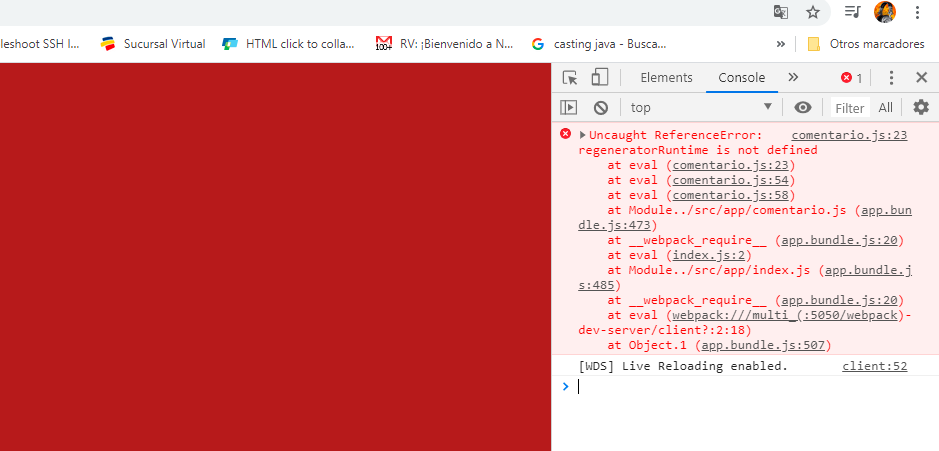
{

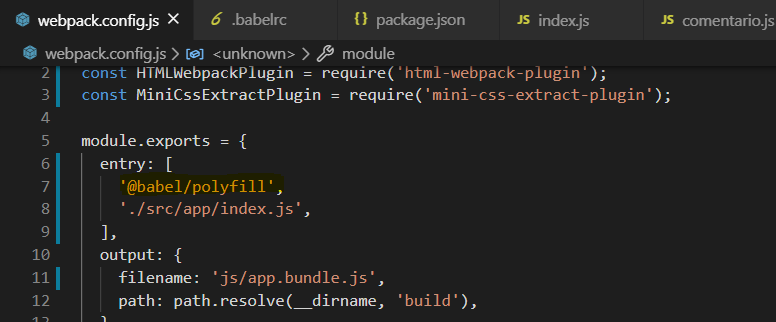
"presets":["@babel/preset-env"]

}

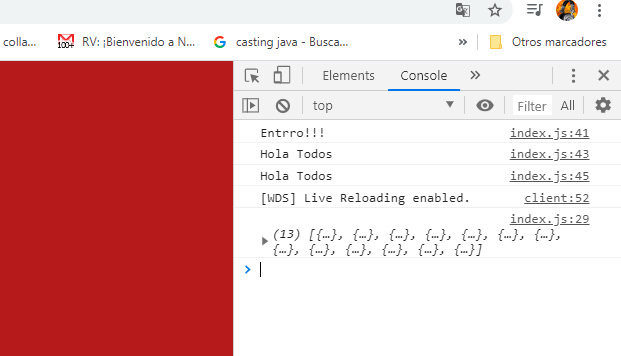


1. Si ejecutamos el proyecto, nos debe aparecer el siguiente error, debido a que se están usando sintaxis nuevas y se debe requerir el módulo @babel/polyfill, por lo cual se debe agregar en el archivo de configuración del de webpack.





1. Al ejecutar nuevamente, veremos que el proyecto funciona correctamente.

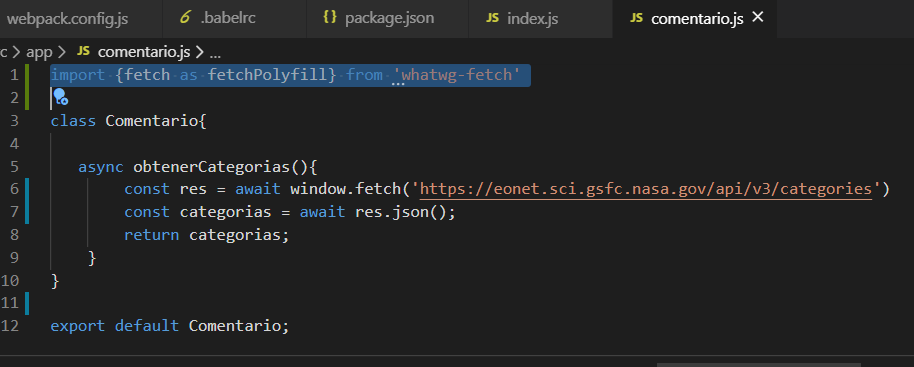


1. Para los errores que presenta IE con fetch, debemos instalar la siguiente dependencia Al ejecutar nuevamente

npm i whatwg-fetch

1. Luego la debemos importar el archivo donde empleemos la función fetch.

import {fetch as fetchPolyfill} from 'whatwg-fetch'



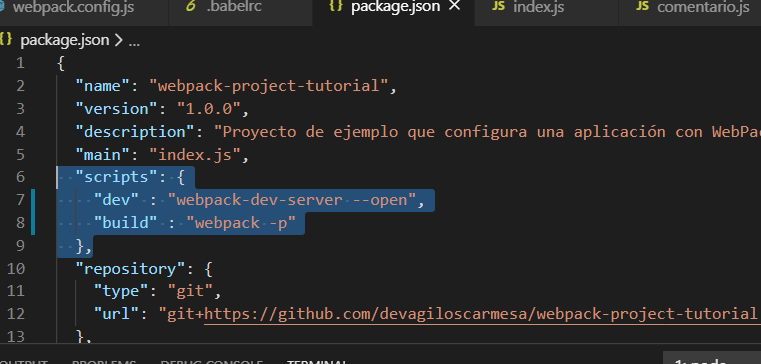
1. Por último, automatizaremos la tarea de ejecutar los comandos que debemos usar constantemente para compilar y desplegar la aplicación. Para ello modificaremos el archivo package.json.

"scripts": {

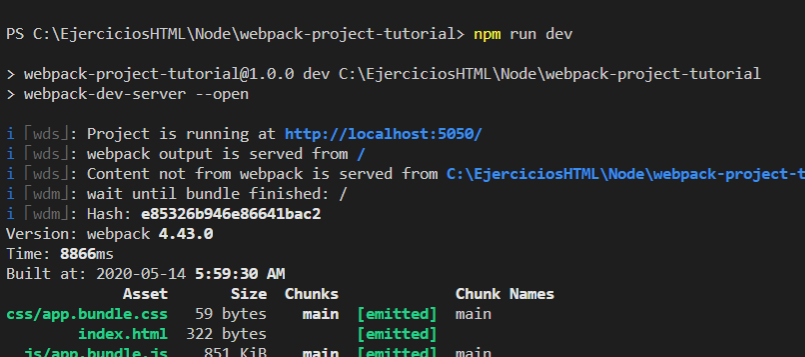
"dev" : "webpack-dev-server --open",

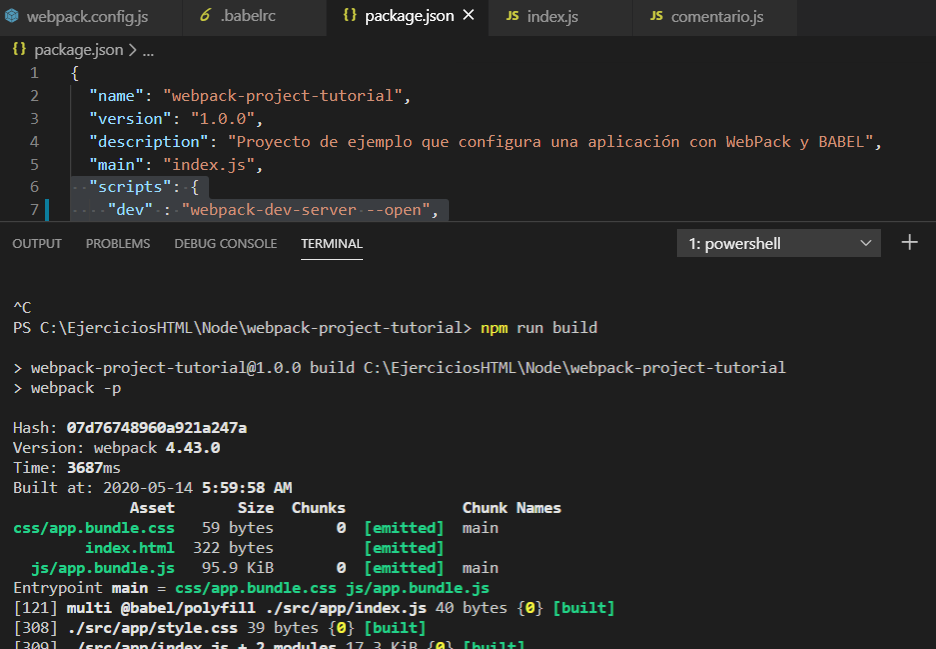
"build" : "webpack -p"

},



Ejecutamos el comando en la consola npm run dev o npm run build.





**SITIOS DE CONSULTA Y PROFUNDIZACIÓN**

* <https://babeljs.io/>
* <https://webpack.js.org/>
* <https://www.npmjs.com/>
* <https://nodejs.org/>
* <https://github.com/devagiloscarmesa/webpack-project-tutorial.git>