Curso de Webpack

# Introducción a webpack

**Webpack** es un empaquetador para Javascript y sus amigos. Convierte módulos con dependencias en archivos estáticos que los navegadores entienden.

Nos permite empaquetar, optimizar los diferentes módulos Javascript y sus dependencia en nuestro proyecto. Es usado en proyectos basados en Javascript como: React, Vue, Angular entre otros.

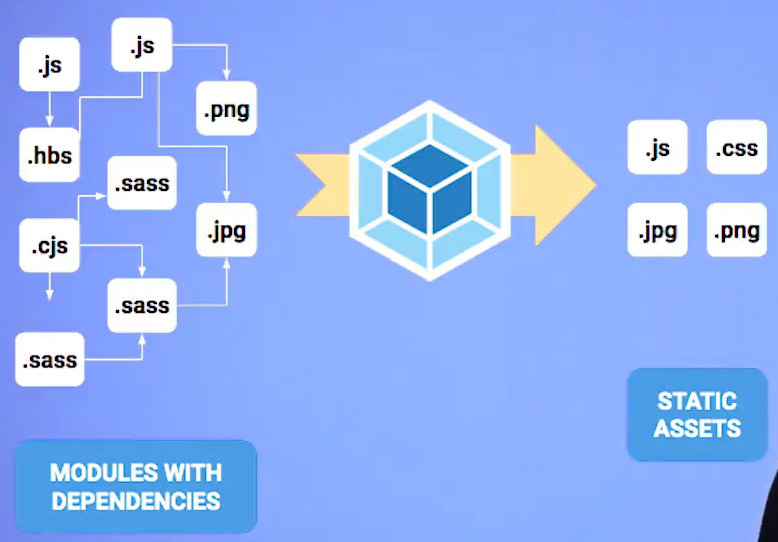
### User Experience

Se logra con una aplicación que:

* Funcione
* Sea rápida
* Cumpla sus necesidades
* Se actualice
* Responda a sus interacciones
* Producto de calidad

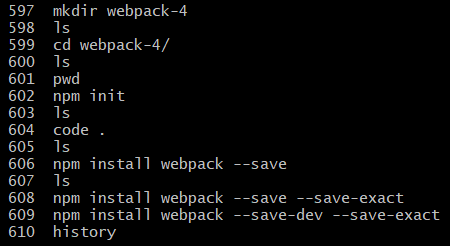
### Developer Experience

* Escribir aplicaciones de manera eficiente.
* Tener un código limpio.
* Aplicar tecnología para resolver sus problemas.
* Tener un conjunto de reglas y convenciones.
* Entorno de desarrollo optimizado en productividad.



Webpack es un conjunto de reglas que vamos a configurar para saber cómo funciona nuestro stack (de archivos), que está incluido en nuestro stack, y cómo manipulamos nuestro stack para que nuestra tecnología pase de la Developer Experience (Modules with Dependencies) a la User Experience (Static Assets).

# Configurando un nuevo proyecto de Javascript

En esta clase el profesor Leonidas Esteban nos guía paso a paso para crear nuestro primer proyecto usando Webpack.

En nuestro proyecto existen dependencias del tipo desarrollo y del tipo core con sus respectivas versiones, por eso usaremos NPM para administrarlas e instalar Webpack que viene a ser una dependencia (de desarrollo) más en nuestro proyecto de Javascript.

Se puede reemplazar install por su shortcut i así:

npm **install** webpack

npm **i** webpack

Las banderas con doble guión como --save se pueden reemplazar con su shortcut usando mayúsculas y sólo un guión así:

npm **install** webpack --save

npm **i** webpack -S

El shortcut de --save-dev es -D

npm install webpack --save-dev

npm **i** webpack -D

Para instalar una versión precisa del paquete se puede agregar a su nombre el número de la versión después del signo @ así:

npm **i** webpack@4.32.2 -D

Este método agregará webpack a las dependencias de desarrollo con el número de versión 4.32.2 pero cuando se descarguen los paquetes se instalará la última versión del paquete debido al prefijo ^

"devDependencies": {

"webpack": "^4.32.2"

}

Para guardar la versión exacta en el package.json se debe agregar la bandera --save-exact y su shortcut sería -E

npm install webpack --save-dev --save-exact

npm i webpack -**D** -**E**

Esto agregará la dependencia sin el prefijo, lo que instalaría la versión exacta indicada en cada instalación.

# Creando nuestro primer bundle con Webpack

Vamos a instalar otra dependencia llamada **webpack-cli**, la API que expone webpack en forma de **CLI (Command Line Interface)** que nos va a permitir interactuar y configurar Webpack desde la terminal.

El comando webpack tiene una bandera llamada --mode que nos permite cambiar entre los modos producción y desarrollo. Recuerda que por defecto nos pone en modo producción si no la especificamos.

**Creacion de bundle con webpack**

Desde la version 4 de webpack requiere el uso de webpack cli el cual se instala como dependencia de desarrollo en nuestro proyecto

**npm install webpack-cli --save-dev**

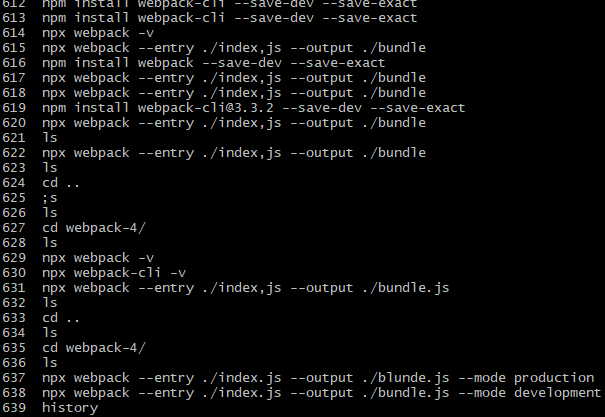
haciendo uso de npx podemos usar webpack como comando en en nuestra consola  
-npx webpack -v (nos arrojara la version instalada)  
\*\*Creacion de archivo e entrada y salida en nuestro webpack  
\*\*  
**npx webpack --entry ./index.js --output ./bundle.js --mode development**

Con este comando le estamos indicando a webpack el archivo en el que trabajaremos nuestro código para que webpack lo procese y luego le indicamos el archivo de salida del bundle el cual se utilizara en el producto final.  
con el flag de “–mode” indicamos si queremos una version de production la cual traera todo el codigo minificado o modo development.

Las **devDependencies** son aquellas que sólo utilizarás tu como desarrollador y que el público no necesitará para correr la página, estas dependencias no serán tomadas en cuenta a la hora de hacer el empaquetado final de tu página, es decir, no se incluirán en la versión de producción. Por ejemplo: ESLint  
   
Por otra parte, las **dependencies** forman parte esencial de tu página y siempre serán incluídas en la versión de producción, pues estas serán necesarias para correr la página. Por ejemplo: React.

npx webpack --entry ./index.js --output ./blunde.js npx webpack --entry ./index.js --output ./blunde.js --mode development npx webpack --entry ./index.js --output ./blunde.js --mode production





# Iniciando un webpack.config

Al ir creciendo nuestra configuración de Webpack iremos agregando cada vez más banderas a nuestros comandos y terminará como una línea gigante en la terminal. ¿Cómo hacemos que esa línea sea muy pequeñita, personalizable y escalable? Por medio de un archivo llamado por defecto **webpack.config.js**.

Este archivo permite importar módulos usando el formato commonJS y recibe por lo menos dos configuraciones básicas, un **entry** y un **output**.

* **Entry Point**: Es la ruta del archivo principal de nuestra aplicación JS a ser procesado por Webpack. Se pueden tener varios Entry Points.
* **Output**: Es la ruta de salida donde va a generar nuestro bundle con todos nuestros archivos especificados como Entry Points empaquetados en uno sólo.



Para ejecutar comandos de webpack hay diferentes caminos 🛤. Anteriormente ejecutamos un comando con NPX para transpilar un archivos .JS y generarlo especificando un OUTPUT, pero esto puede ser en la práctica algo que quite más tiempo y engorroso.

Para ello se puede crear un archivo de configuración de webpack. El nombre de este archivo en teoría puede ser cualquiera pero se recomienda llamarlo webpack.config.js.

Dentro de este archivo se especificarán las acciones que queramos que webpack haga por nosotros. El archivo de ejemplo es este:

**const** path = require(‘path’) /\*este es un modulo de node y se encarga de la gestion de rutas\*/

/\*Para exportar un modulo, se debe hacer de la forma de commonJS\*/

module.exports = {

entry: ‘./index.js’, /\*El archivo que queremos transpilar \*/

mode: ‘development’, /\* modo de ejecución \*/

output:{ /\*Configuraciones para el archivo final que generará webpack \*/

path: path.resolve(\_\_dirname), /\* La ruta en donde quedará el archivo final \*/

filename: ‘bundle.js’ /\*El nombre con el que se creará el archivo final \*/

}

}

Una vez se tenga esta configuración ⚙️ se puede llamar de dos maneras ✌🏽:

* con npx: Se puede llamar con este comando para que webpack tome el archivo de configuración y haga su proceso con base en este. El comando es

npx webpack

* con npm: Dentro del archivo .JSON, dentro del objeto scripts se puede configurar un comando para ser llamado directamente por npm. El comando que se añadirá es: build, que a su vez llamará a webpack

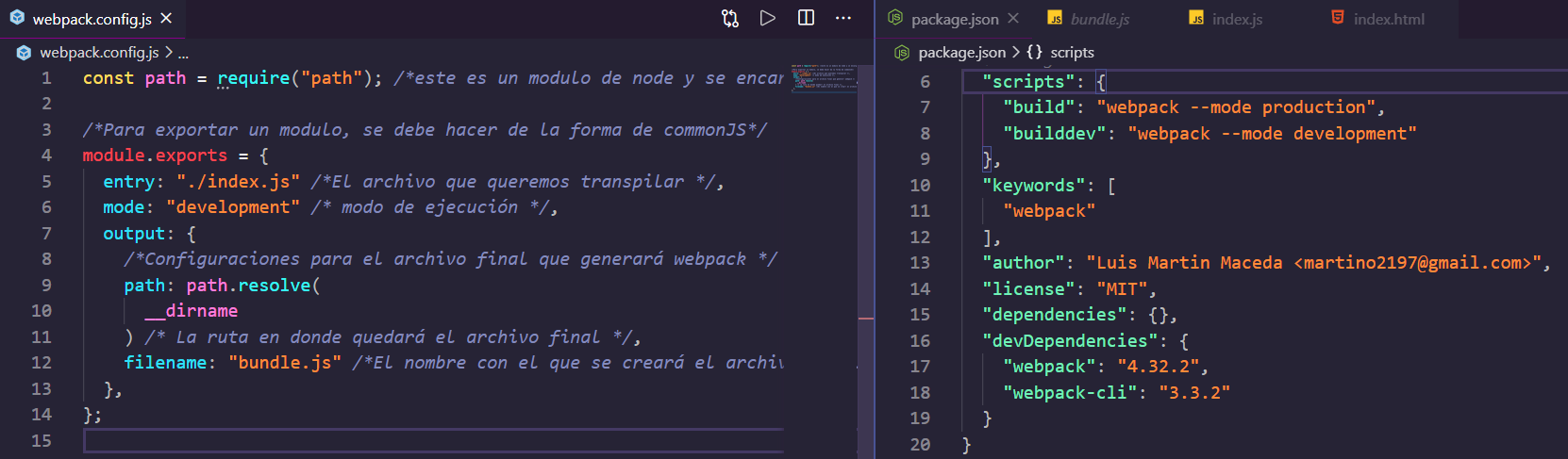
“scripts”:{

“**build”:** “webpack” //**build** sera el nombre que usaremos en consola

}

Para ejecutarlo basta con escribir en la consola 🤩:

npm run webpack



# Cargando configuraciones por defecto y personalizadas

Por medio del uso de la bandera --config podemos especificar un archivo de configuración externo con el nombre que queramos en lugar del nombre por defecto **webpack.config.js**.

Resumen de clase:

En esta clase se agregó una configuración de webpack diferentes para un build externo. para eso se agregó.  
Dentro de la carpeta **externals** un **webpack.config.js**  
con la siguiente configuración

**const** path = require('path')

module.exports = {

entry: path.resolve(\_\_dirname, 'index.js'),

mode: 'development',

output: {

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

filename: 'bundle.js'

}

}

donde le indicamos que nuestro bundle se guarde dentro de la carpeta dist.

Tambien se agregaron 3 cmandos en npm scripts para probar las diferentes ejecuciones.

"scripts": {

"build": "webpack --entry ./index.js --output ./bundle.js --mode development",

"build:local": "webpack --mode production",

"build:external": "webpack --config ./external/webpack.config.js"

},

Usualmente yo uso dos archivos de configuración que fue la forma que encontré en la documentación

* webpack.dev.js
* webpack.prod.js

Y los scripts para trabajar con uno u otro archivo de configuración los creo como dice Leonidas usando el flag --config

**Apuntes**

* Dentro de dist van los compilados de webpack (es una convencion de desarrolladores)
* Con --config le digo a webpack-cli donde esta el archivo de configuracion
* Es posible crear diferentes archivos config para webpack:

"scripts": {

"build:local": "webpack --entry ./index.js --output ./bundle.js --mode development",

"build:production": "webpack --mode production",

"build:external": "webpack --config ./external/webpack.config.js"

# Múltiples puntos de entrada



Ya que se está usando path para las rutas lo recomendable sería separar las carpetas por comas

**const** path = require('path');

module.exports = {

mode: 'development',

entry: {

home: path.resolve(\_\_dirname, 'src', 'javascripts', 'index.js'),

precios: path.resolve(\_\_dirname, 'src', 'javascripts', 'precios'),

contacto: path.resolve(\_\_dirname, 'src', 'javascripts', 'contacto')

},

output: {

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

filename: 'bundle-[name].js'

}

};

porque justamente es el módulo “path” de npm el que transpila la dirección que le pasamos por parámetro al método “resolve” a los diferentes sistemas operativos, ya que algunos utilizan “/”, otros “”, otros “//” y así. Ese es el sentido de separar las carpetas por comas.

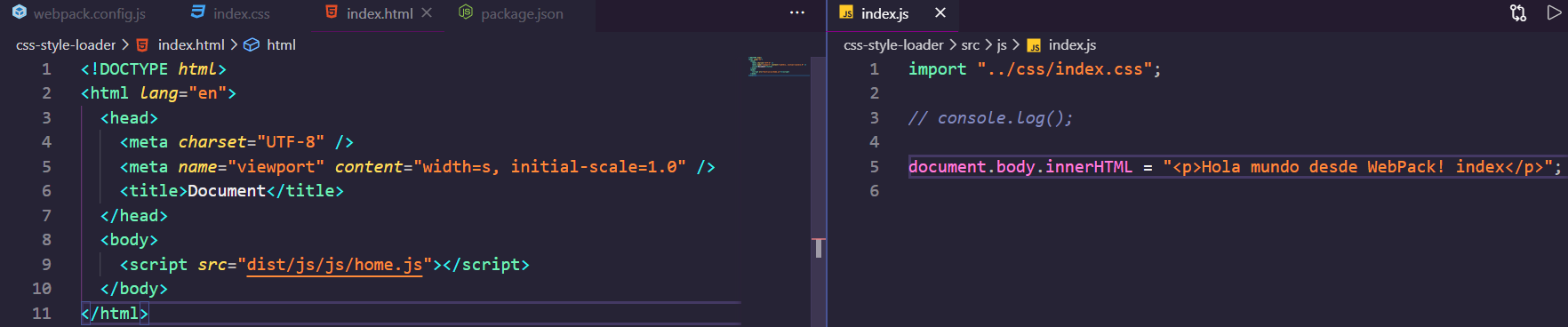
# Manejo de assets con Loaders

**Los Loaders** son la funcionalidad que nos da Webpack para interpretar tipos de archivos no soportados de forma nativa por Javascript.

**style-loader** sirve para inyectar un tag style (el CSS) al DOM de nuestro HTML, mientras que **css-loader** sólo sirve para interpretar archivos CSS.

npm install --save-dev --save-exact css-loader

npm install --save-dev --save-exact style-loader



<https://webpack.js.org/concepts/loaders/#configuration>

# Introducción al uso de plugins

**Los Plugins** sirven para extender las capacidades de webpack y dar más poder a los loaders.

npm install mini-css-extract-plugin html-webpack-plugin --save-dev --save-exact

**Resumen de clase:**

En esta clase se vio una introduccion al uso de plugin en webpack con dos muy utiles.

**MiniCssExtractPlugin**  
npm install mini-css-extract-plugin --save-dev

Este plugin nos generara nuestro archivo de estilos de forma separada.  
**HtmlWebpackPlugin**  
npm install html-webpack-plugin --save-dev

Este archivo nos generara un estructura html con los archivos js y css de dist ya importados

webpack.config.js:

**const** path = require('path');

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

**const** HtmlWepackPlugin = require('html-webpack-plugin');

module.exports = {

entry: path.resolve(\_\_dirname, 'src/js/index.js'),

mode: 'development',

output: {

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

filename: 'js/index.js'

},

module: {

rules: [

{

test: /\.css$/,

use: [

{

loader: MiniCssExtractPlugin.loader

},

'css-loader'

]

}

]

},

plugins: [

**new** MiniCssExtractPlugin({

filename: 'css/[name].css'

}),

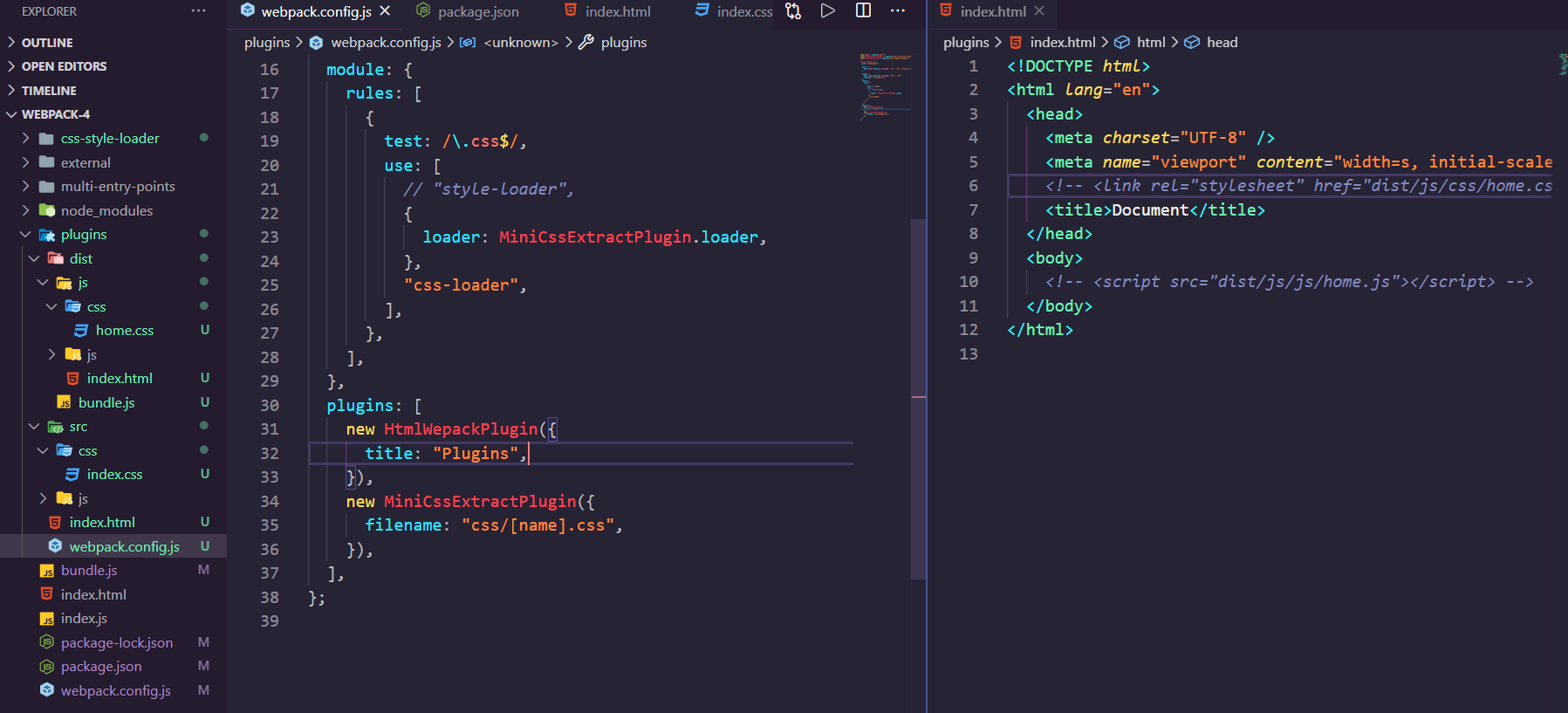
**new** HtmlWepackPlugin({

title : 'Nuevo titulo con el plugin htmlwebpackplugin'

})

]

}



# Servidor de desarrollo

npm run build:dev -- -w

## Webpack Dev Server o Servidor de desarrollo

[webpack-dev-server](https://github.com/webpack/webpack-dev-server) es un [CLI (command line interface)](https://www.w3schools.com/whatis/whatis_cli.asp) que se puede usar para hacer un entorno de desarrollo de manera sencilla, ya que de manera nativa recargará los archivos que se modifiquen en tiempo real.  
.  
webpack-dev-server usará [webpack](https://webpack.js.org/) con un servidor de desarrollo cuya funcionalidad principal es la recarga en vivo. Esto debe usarse para **desarrollo solamente** porque estará escuchando los cambios que se realicen en tiempo real, lo que es una carga innecesaria en memoria, porque los archivos en servidores de producción no deberían cambiar seguido, haciendo insostenible el uso intensivo de la aplicación.  
.  
webpack-dev-server utiliza [webpack-dev-middleware](https://github.com/webpack/webpack-dev-middleware) por debajo, mismo que proporciona un acceso rápido en memoria a los assets de webpack.

Les dejo un link para complementar la excelente clase del profesor: <https://youtu.be/vF2emKbaP4M>

Webpack 4 es un empaquetador de módulos, muy utilizado en archivos del Frontend y necesario en todos los frameworks de Javascript como React, Angular, y Vue.js y otros desarrollos de Javascript en el Frontend.

CURSO DE NODEJS  
<https://youtu.be/BhvLIzVL8_o>

APLICACION DEL CLIMA CON JAVASCRIPT Y WEBPACK  
<https://youtu.be/yxT6ylPM7uM>

CLON DE GITHUB CON WEBPACK  
<https://youtu.be/I4kJKnu6_jc>

VARIABLES DE ENTORNO EN NODEJS  
<https://youtu.be/U6st9-lNUyY>

RED SOCIAL NODEJS Y MONGODB  
<https://youtu.be/TqC3e8nBycg>

JAVASCRIPT ORIENTADO A OBJETOS CON WEBPACK  
<https://youtu.be/nqre9kKFRpc>

CURSO DE GIT:  
<https://youtu.be/HiXLkL42tMU>

VIDEO DE VSCODE:  
<https://youtu.be/zbycB-Yetb0>

VISITA MI SITIO WEB PARA MÁS CURSOS Y TUTORIALES  
[http://www.faztweb.com](http://www.faztweb.com/)  
[http://blog.faztweb.com](http://blog.faztweb.com/)

FAZT CODE - YOUTUBE:  
<https://www.youtube.com/channel/UCMn2>…

TWITTER  
<http://twitter.com/fazttech>

INSTAGRAM  
<https://www.instagram.com/fazttech/>

FACEBOOK  
<http://facebook.com/fazttech>

COMUNIDAD EN DISCORD  
<https://discord.gg/37PHuNw>

#webpack #javascript #frontend

# Hot Module Replacement

## Configuración del devServer en webpack.config

Para configurar el webpack-dev-server puedes agregar un objeto llamado [devServer](https://webpack.js.org/configuration/dev-server/" \t "_blank) dentro de tu webpack.

module.exports = {

//...

devServer: {

// webpack-dev-server configuration

}

};

En el devServer podemos configurar cosas como:

* hot: Activa el HotModuleReplacementPlugin que muestra los cambios sin actualizar la página
* port: Al crear un servidor de prueba con webpack-dev-server éste tomará por defecto el puerto 8080, acá podrás cambiar el puerto.
* open: Abrirá el navegador al iniciar el servidor
* allowedHosts: Dominios que permitirá el servidor utilizar
* [Etc…](https://webpack.js.org/configuration/dev-server/)

Configuraciones del devServer

## HotModuleReplacementPlugin

Hot Module Replacement (HMR) es un plugin de Webpack que permite intercambiar, agregar o eliminar [módulos](https://webpack.js.org/concepts/modules/) en tiempo de ejecución, sin una recarga completa de la página.  
.  
Para activarlo lo tienes que importar desde webpack, agregarlo como plugin y en el devServer agregar la propiedad hot

**const** webpack = require('webpack');

module.exports = {

//...

devServer: {

hot: **true**

}

plugins: [

**new** webpack.HotModuleReplacementPlugin()

]

};

HMR puede acelerar significativamente el desarrollo de las siguientes formas:

* Conservando el estado de la aplicación que se pierde durante una recarga.
* Ahorre un valioso tiempo de desarrollo actualizando solo lo que ha cambiado.
* Actualice al instante el navegador cuando se realicen modificaciones a CSS / JS en el código fuente, que es casi comparable a cambiar los estilos directamente en las herramientas de desarrollo del navegador.

Trabajando con loaders y plugins

# Soporte de Javascript moderno

Javascript es un lenguaje moderno en evolución, siempre agregando nuevas funciones. El problema es que al ser interpretado en el navegador, no tenemos control sobre que versión de Javascript soportan y por lo tanto que funciones.

Para poder usar Javascript moderno y tener una buena **Developer Experience** sin afectar la **User Experience**, existe **Babel**. Babel transpila nuestro código moderno de Javascript a una una versión que todos los navegadores pueden entender.

npm install --save-dev --save-exact @babel/core babel-loader @babel/preset-env

## Soporte de Javascript moderno con babel

Javascript es un lenguaje en constante evolución, cada año se encuentra agragando nuevas funcionalidades.  
.  
Una de las mayores ventajas y desventajas que tenemos es que javascript se corre en el navegador, lo que significa uso menor de recursos de parte de nuestros servidores, pero también que los navegadores se actualizan más lento que los nuevos releases de las versiones, además de que existen diferentes navegadores que soportan [diferentes funcionalidades](https://www.w3schools.com/js/js_versions.asp).  
.  
Para poder tener un mejor **developer experience** (que los desarrolladores programen más rápido) con las funcionalidades modernas de JavaScript es necesario utilizar un [transpilador de código](https://ingenieriadesoftware.es/diferencia-transpilacion-compilacion/), siendo el más popular [babel](https://github.com/babel/babel).  
.  
El papel de **babel** será el de **transpilar** (convertir y adaptar código) escrito en estándares modernos (normalmente superior a ecmaScript 2015) a estándares que [soporten la mayoría de los navegadores](https://www.w3schools.com/js/js_versions.asp), de esta manera el desarrollador se preocupa por porgramar de manera cómoda y webpack con babel lo convierten en lo que sea mejor para los navegadores.  
.  
Para hacer uso de **Babel** en webpack, como éste tiene un loader (babel-loader), tenemos que instalar @babel/core (que tiene la funcionalidad básica de babel) y babel-loader e instanciar el loader dentro de nuestras rules de la siguiente manera:

module.exports = {

//

module: {

rules: [

{

test: /\.js$/,

use: 'babel-loader', // Intercepta los archivos de js y transpila en versiones más antiguas que entiendan la mayoría de los navegadores

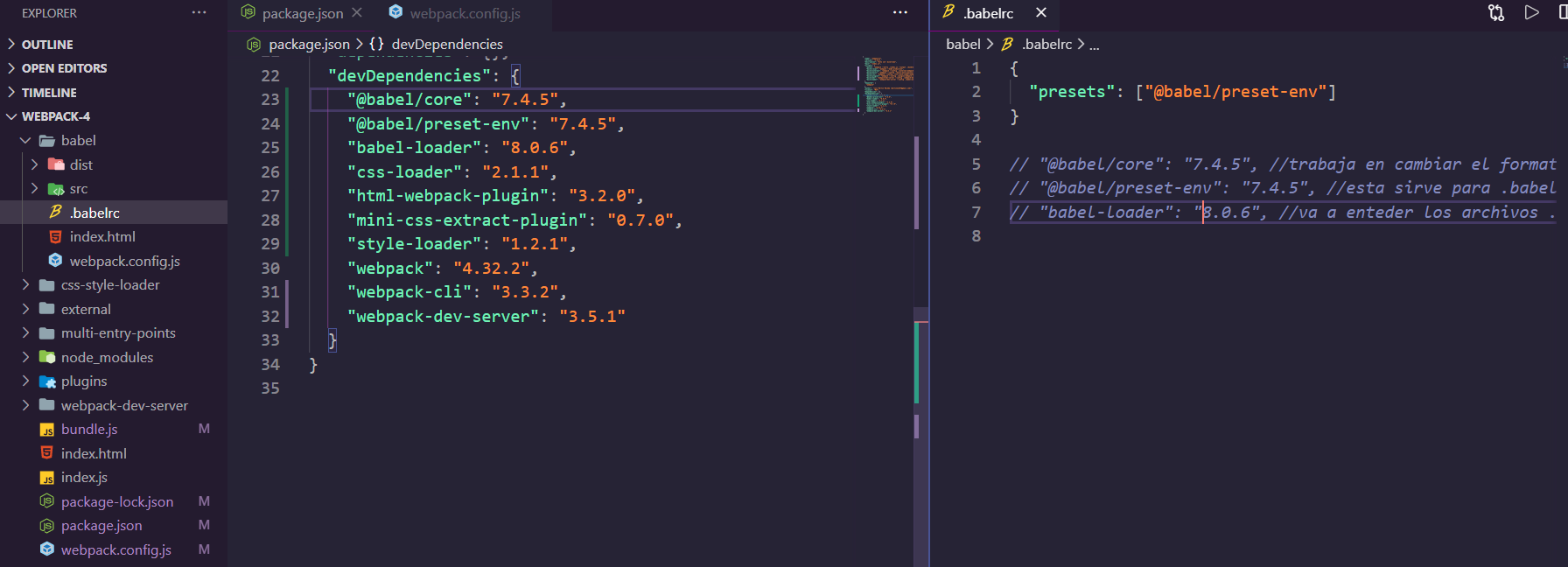
exclude: /node\_modules/,

},

},

};

Babel requerirá de configuración, ésta puede ser declarada dentro del webpack, pero se recomienda hacerlo dentro de un archivo llamado [.babelrc](https://babeljs.io/docs/en/config-files) que leerá babel a la hora de ser configurado por webpack.

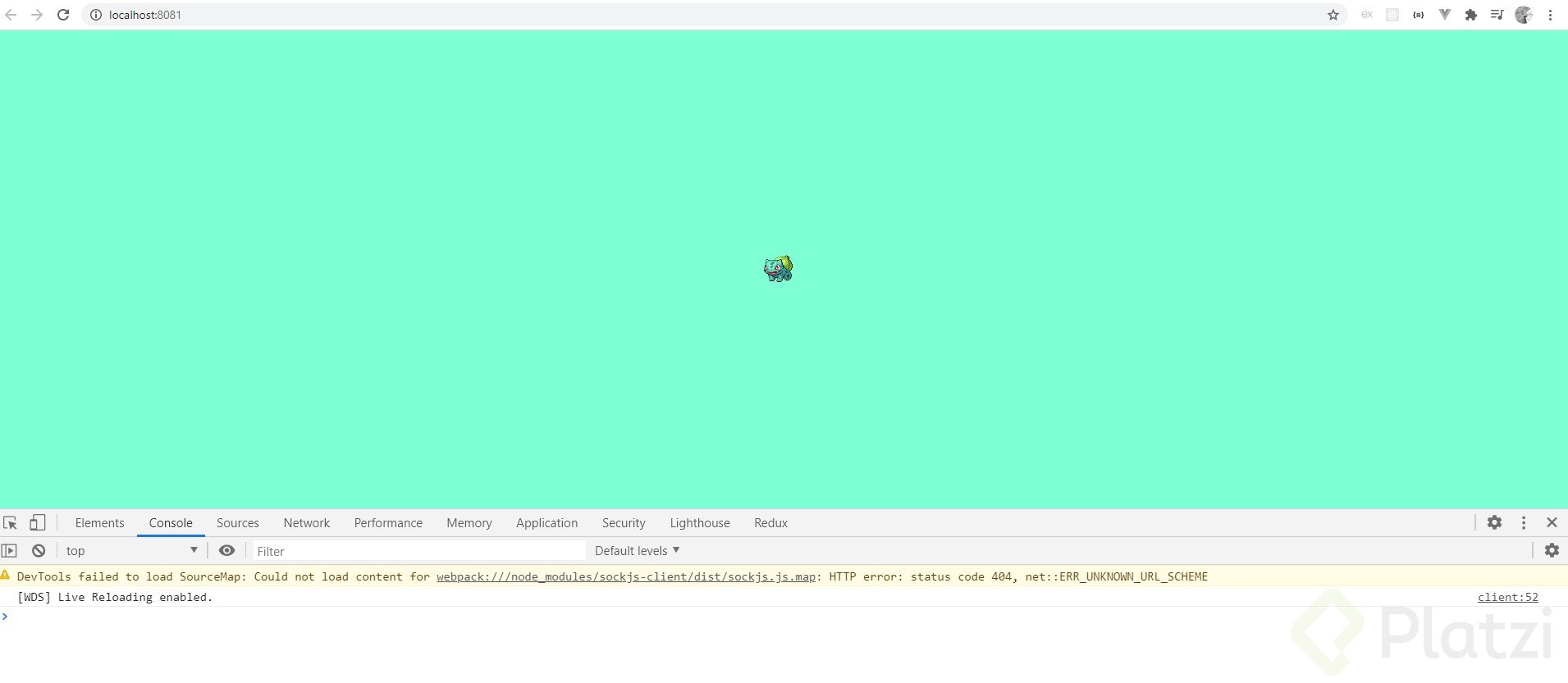
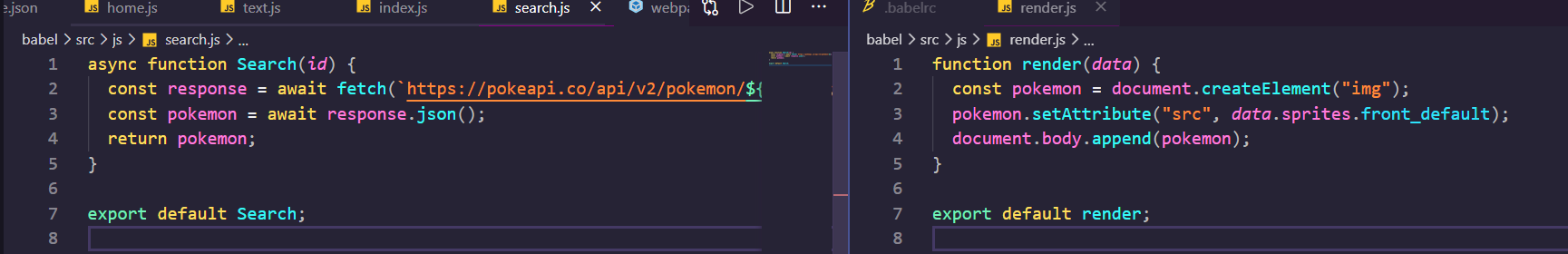


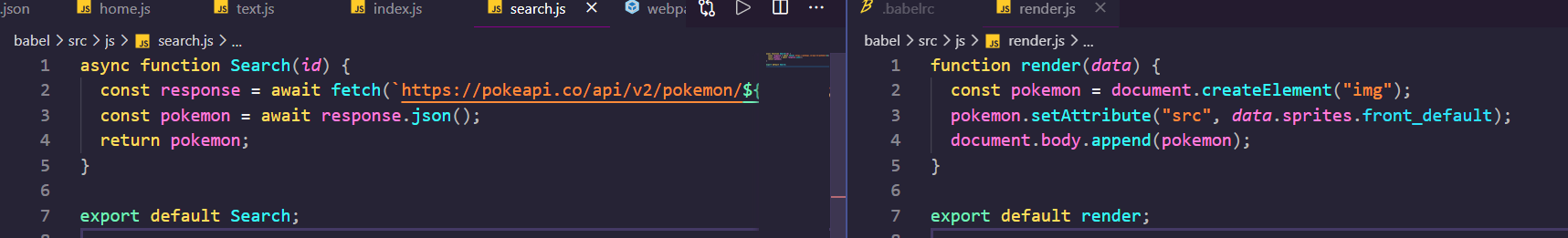
# Trabajando con Javascript moderno

npm install --save-dev --save-exact @babel/plugin-transform-runtime

npm install --save --save-exact @babel/runtime







# Soporte de JSX (React)

**JSX** es un lenguaje de templates para React que permite definir componentes con un código muy similar al HTML.

No existe navegador que entienda JSX porque no es un estándar, es algo específico de React. Afortunadamente Babel puede transpilar el código JSX de nuestros archivos JS a código que el navegador.

npm install --save-dev @babel/preset-react --save-exact

npm install --save --save-exact react react-dom

**¿Cómo instalar React.js con Webpack**

Genial, tu y yo ya tenemos la base de Webpack y ahora instalaremos React, esto es lo que haremos

1. Instalar las siguientes dependencias:

npm install --save-dev --save-exact @babel/preset-react

npm install --save --save-exact react

npm install react-dom --save --save-exact

//Nota: El orden de los flags no importa

1. Actualizar el archivo de .babelrc (el archivo que tiene tu configuración de babel):

"presets": [

"@babel/preset-env",

"@babel/preset-react"

]

React necesita un elemento container en el HTML, para ello modificaremos nuestro HTML hecho automaticamente con webpack

1. Actualizar en webpack.config.js el plugin de HTML

plugins: [

**new** webpack.HotModuleReplacementPlugin(),

**new** HtmlWebpackPlugin({

title: "plugins",

minify: {

collapseWhitespace: **true**

},

template: path.resolve( \_\_dirname, "index.html" )

})

]

1. Y añadiremos el elemento container que llamaremos en el index.js de React.

<!DOCTYPE html>

<html>

<head>

<title></title>

<meta charset="utf-8">

</head>

<body>

<div id="root"></div>

</body>

</html>

1. ¡Y voilá! *Haz que algo que marque la diferencia suceda con tu App en React.js*

Index.js

//Gracias a Webpack es que es posible hacer esto

**import** React from "react";

**import** {render} from "react-dom";

**import** App from "./components/App.js";

**import** "../css/index.css";

render(<App />, document.getElementById("root"))

App.js

**import** React **from** "react";

**function** **App**(props) {

**return**(

<section className="App">

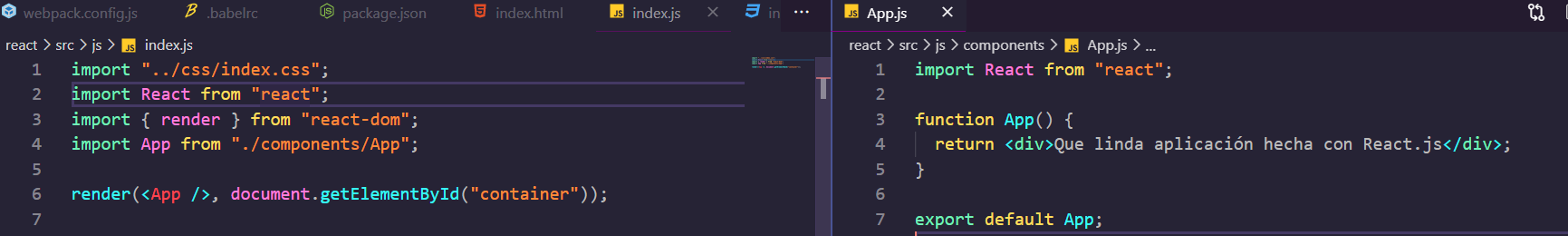
<h1>App Testing</h1>

</section>

)

}

**export** **default** App;



# Trabajando con React

Destructuring

**function** **Loader**(props) {

**return** (<li>{props.data.name}</li>);

}

**function** **Loader**({data}) {

**return** (<li>{data.name}</li>);

}

**function** **Loader**({data:{name}}) {

**return** (<li>{name}</li>);

}

## Aqui les dejare toda la configuracion del proyecto por si se perdieron:

.

<h1>1- En el webpack.config.js</h1>

**const** path = require('path')

**const** MiniCSSExtractPlugin = require('mini-css-extract-plugin')

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

**const** webpack = require('webpack')

module.exports = {

entry: {

home: path.resolve(\_\_dirname,'src/js/index.js'),

},

mode: 'development',

output: {

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

filename: 'js/[name].js'

},

devServer: {

hot: **true**,

open: **true**,

port: 9000,

},

module: {

rules: [

{

test: /\.js$/,

use: 'babel-loader',

exclude: /node\_modules/,

},

{

test: /\.css$/,

use: [

'style-loader',

'css-loader'

]

}

]

},

plugins: [

**new** webpack.HotModuleReplacementPlugin(),

**new** HtmlWebpackPlugin({

title: 'webpack-dev-server',

template: path.resolve(\_\_dirname, 'index.html')

}),

]

}

<h1>2-Index.html</h1>

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<section id="container"></section>

</body>

</html>

<h1>3- .babelrc</h1>

{

"plugins": [

"@babel/plugin-transform-runtime"

],

"presets": [

"@babel/preset-env",

"@babel/preset-react"

]

}

<h1>4- src/index.js</h1>

**import** '../css/index.css'

**import** React **from** 'react'

**import** { render } **from** 'react-dom'

**import** App **from** './components/app'

render(<App />, document.getElementById('container'))

<h1>5- src/components/app.js</h1>

**import** React **from** 'react'

**function** **App**() {

**return** (

<div>

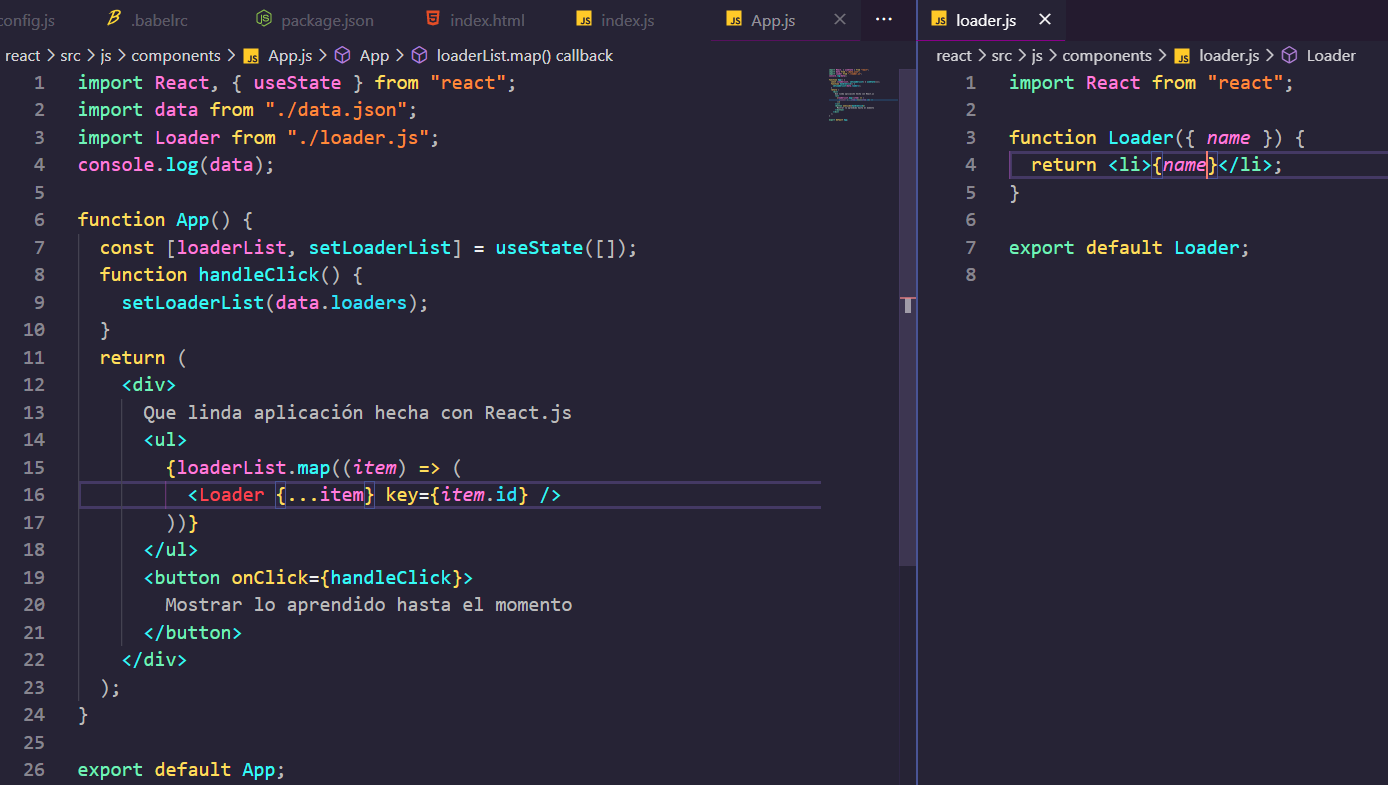
que linda aplicación hecha en React.js

</div>

)

}

**export** **default** App



# Soporte imágenes, fuentes y videos

Para soportar la importación de archivos binarios en nuestro código Javascript cómo lo son: fuentes, imágenes y videos, podemos usar **url-loader**.

url-loader transforma archivos a un cadena de texto base64 para que carguen dentro de nuestros archivos Javascript y así ahorrarnos un request al servidor por cada archivo transformado.

Debemos tomar en cuenta que sólo nos conviene convertir archivos pequeños, ya que archivos muy grandes podrían hacer nuestro archivo bundle muy pesado. Es por esto que la opción **limit** del url-loader sirve para asignar el peso máximo que un archivo puede tener para ser transformado en base64.

No olvides instalar **file-loader** junto con **url-loader** ya que cuando se sobrepasa el limite establecido en la opción limit y el archivo no pueda ser transformado a base64, **url-loader** hará uso del **file-loader** para insertar un nombre y ruta de archivo en el lugar correspondiente.

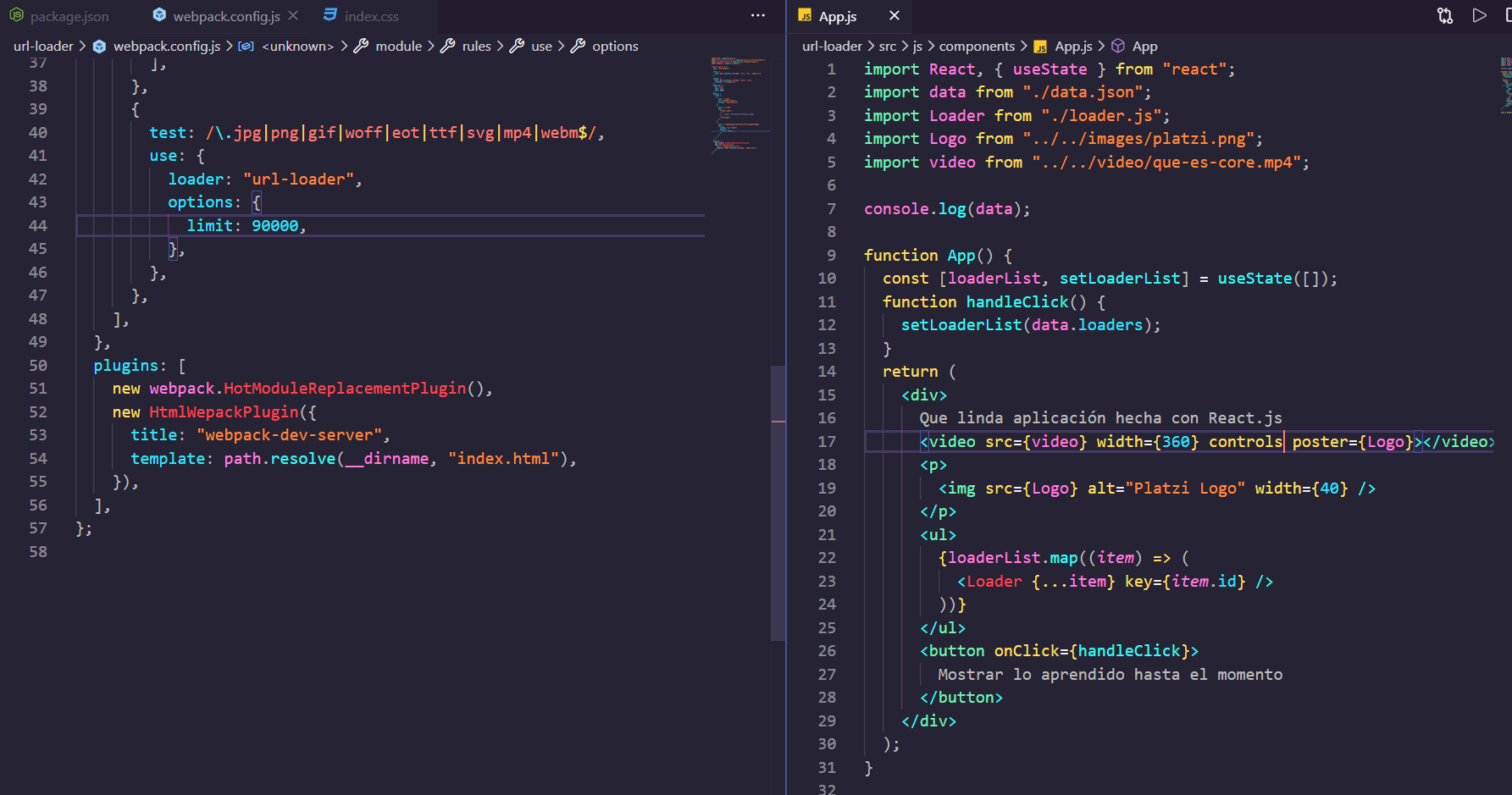
npm install --save-dev --save-exact url-loader

npm install --save-dev --save-exact file-loader

Hola, base64 es sistema que codifica archivos utilizando la mayor potencia de los bits (2) que es 64, se usa en imagenes pequeñas de 100kb aprox. Con esto ayudamos al performance de un aplicación; ya que, no vamos a hacer peticiones adicionales al servirdor. Más claramente, esta imagen esta encodeada en el bundle de la App y con esto evitamos hacer peticiones HTTPRequest al server.

El limit es para que los archivos no excedan el limite de su peso en bytes para transformalos en Base64.

Aquí hay un link para convertir imagenes en Base64 y deseas utilizar imagenes más grandes los mejor es utilizar la imagen normalmente.  
<https://www.base64-image.de/>



# Estilos con preprocesadores

npm install sass-loader stylus-loader less-loader postcss-loader --save-dev --save-exact

npm install stylus less node-sass --save-dev --save-exact

npm install --save-dev --save-exact postcss-nested@4.2.3

Es una práctica común usar preprocesadores de CSS como: Sass, Less, Stylus y hasta PostCSS. Webpack permite integrar estos preprocesadores en su configuración a través de loaders, sólo ten cuidado con las **peerDependencies** que son dependencias que el loader espera estén instaladas previamente, como el caso de **stylus** para **stylus-loader**.

#Mi resumen

Tutorial: Cómo trabajar con preprocesadores de CSS en Webpack

1. **Necesitas instalar los loaders de los preprocesadores:**

npm **install** sass-loader

npm **install** stylus-loader

npm **install** **less**-loader

npm **install** postcss-loader

--save-dev --save-exact

//Versión Corta

npm **install** sass-loader stylus-loader **less**-loader postcss-loader --save-dev --save-exact

1. Antes de configurar los preprocesadores en webpack necesitas tenerlos previamente instalado en tu computador, tenerlos en tus Peer dependencies.

Instalando preprocesadores de CSS

npm **install** sass

npm **install** node-sass

npm **install** stylus

npm **install** **less**

--save-dev --save-exact

//**Short** **version**

npm **install** sass stylus **less** --save-dev --save-exact

1. Necesitas añadir los loaders en tus archivo de webpack.config.js.

{

test: /\.scss$/,

use: [

"style-loader",

"css-loader",

"sass-loader"

]

},

{

test: /\.less$/,

use: [

"style-loader",

"css-loader",

"less-loader"

]

},

{

test: /\.styl$/,

use: [

"style-loader",

"css-loader",

"stylus-loader"

]

}

1. ¡Y voilá! Pon a correr tu webpack config y crea unos diseños inspiradores para tu aplicación web.

En mi caso:

npm run build:precss

En caso de PostCSS …  
#Mi resumen

**Tutorial: Cómo trabajar con POST CSS en Webpack**

1. Necesitas instalar el loader de PostCSS:

npm install postcss-loader --save-dev --save-exact

1. Configura el loader de POST CSS en webpack.config.js

*Nota:*

* Los archivos de post css son .css
* Primero aplica css loader y despues postcss loader

test: /\.css$/,

use: [

"style-loader",

{

loader:"css-loader",

options: {

importLoaders: 1

}

},

"postcss-loader"

1. PostCSS trabaja con plugins, instala lo que usarás.

npm install --save-dev --save-exact postcss-nested

1. Añade los plugins usados en postcss.config.js

module.exports = {

plugins: {

"postcss-nested": {

}

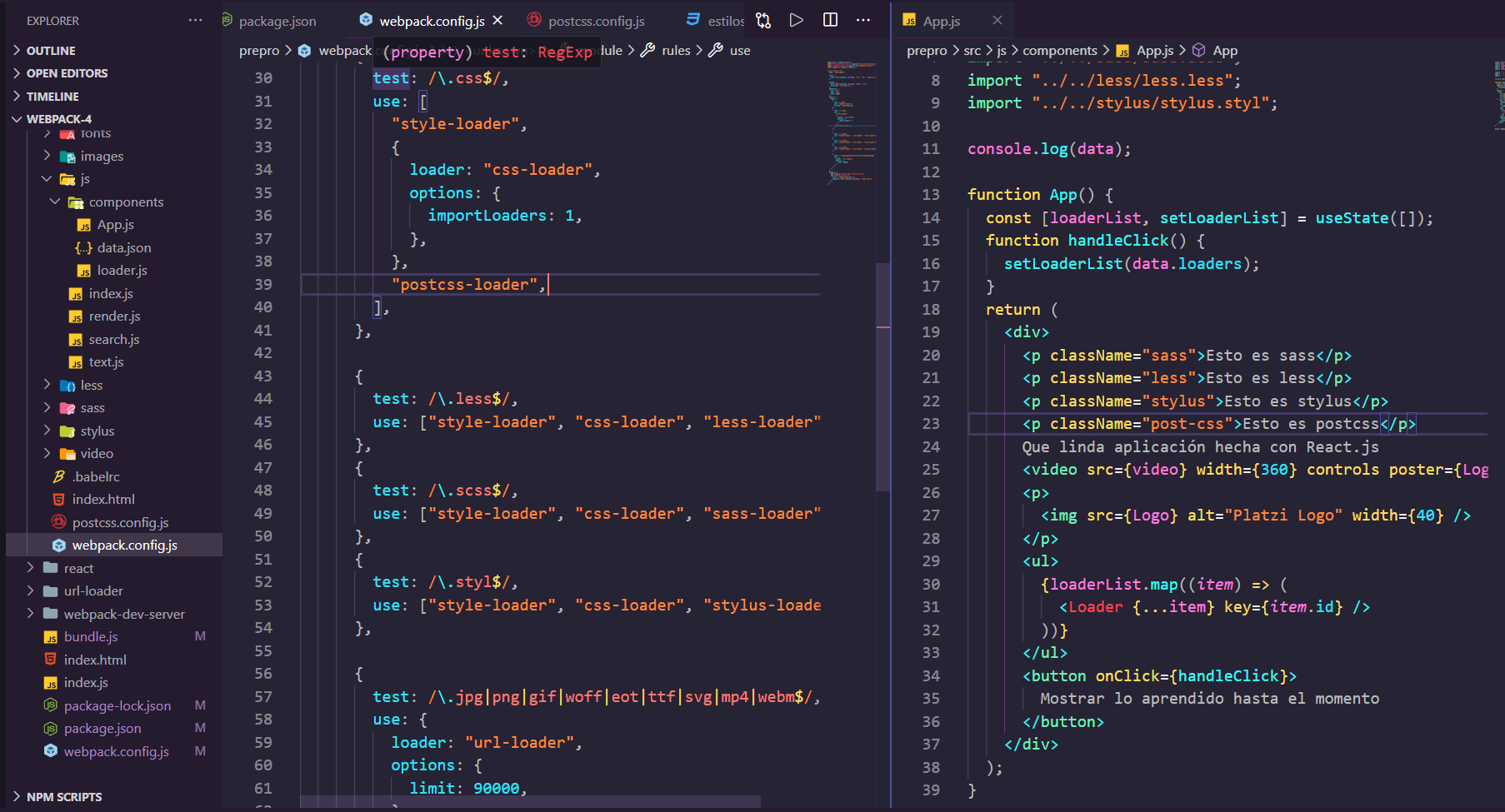
}

}

1. ¡Y voilá! Pon a correr tu webpack config y crea unos diseños inspiradores para tu aplicación web.

En mi caso:

npm run build:precss



Conceptos Avanzados de Webpack

# Evitar código duplicado

<https://webpack.js.org/plugins/split-chunks-plugin/>  
Es útil dividir nuestro código en diversos archivos y a veces enteros proyectos, pero no queremos cargar nuestra aplicación de más multiplicando el peso de alguna dependencia al utilizarla en diferentes partes de la aplicación, para eso utilizamos el módulo de optimización con splitChunks en webpack..

module.exports = {

entry:{

home: path.resolve(\_\_dirname, './src/js/index.js'),

contact: path.resolve(\_\_dirname, './src/js/contact.js'),

},

devServer:{...},

output:{...},

module:{...},

plugins: [...],

optimization:{

splitChunks:{ // permite dividir el peso en diferentes archivos para evitar duplicación

chunks:'all',

minSize:0,

name:'commons',

}

},

}

Resumen de la clase

1)) Agregar lo siguiente al archivo **webpack.config.js**:

mode: 'production',

optimization: {

splitChunks: {

chunks: "all",

minSize: 0,

name: "commons"

}

}

2)) Llamar al script **commons.js** generado en los archivos html:

<script src="dist/js/commons.js"></script>

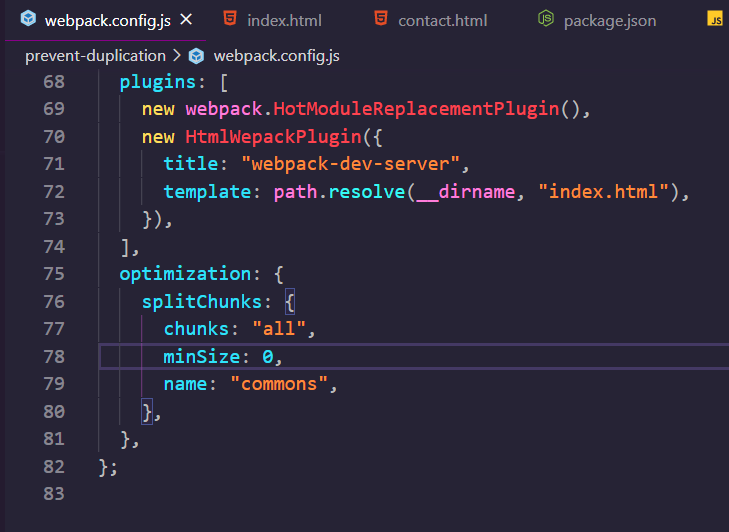
<script src="dist/js/index.js"></script>

La propiedad chunk de splitChunks puede recibir cualquiera de estos tres parámetros:

* ‘initial’: indica a webpack que optimice solo los módulos que son importados de forma no dinámica.
* async: webpack optimizará solo aquellos que son cargados de forma no-dinámica.
* all: indica a webpack que optimice, lo mejor que pueda, los módulos sin importar si son cargados de forma dinámica o no-dinámica.

Comparto esta lectura que lo explica más a detalle y con un ejemplo practico.

* [Webpack 4 — Mysterious SplitChunks Plugin](https://medium.com/dailyjs/webpack-4-splitchunks-plugin-d9fbbe091fd0)



# Añadiendo un Dynamic Link Library

Mientras más librerías agregamos más lento se empiezan a volver nuestros builds, arruinando así la **Developer Experience**. Por suerte podemos crear una (o varias) Dynamic Link Library para acortar estos tiempos.

Una **Dynamic Link Library (DLL)** es un conjunto de librerías comunes que no cambian frecuentemente por lo que se hace un build por adelantado de las mismas para no re-empaquetar cada vez que hacemos build de nuestra aplicación.

Beneficiando tanto la **Developer Experience** como la **User Experience** ya que el caché del navegador va a mantener una copia que solo va a cambiar cuando nosotros agreguemos o quitemos alguna dependencia, ahorrando así valiosos requests al servidor.

Principio subyacente de la clase.

1. Mientras más dependencias más lento será tu build de production.

¿Qué acciones clave extraigo de esta clase ?

*Para hacer el build más rápido vamos a aplicar la técnica de****dynamic link library****.*

*“Una Dynamic Link Library (DLL) es un conjunto de librerías comunes que no cambian frecuentemente por lo que se hace un build por adelantado de las mismas para no re-empaquetar cada vez que hacemos build de nuestra aplicación.”*.

#Tutorial:  
Cómo crear un Dynamic Link Library para acortar el tiempo de carga de librerias. (Estamos creando un webpack config super optimizado para produccion).

1. Creamos los siguientes archivos:

* webpack.config.js (el webpack de toda la vida).
* webpack.dll.config.js (el webpack para las librerias).

1. Colocamos las dependencias core que se repite su uso en nuestra app en el entry de nuestro webpack.dll.config.js. para guardarla en cache del navegador y precargarlas.

webpack.dll.config.js

**entry**: {

modules: [

"react",

"react-dom"

]

},

1. Añadimos plugin de DLL en nuestro webpack.dll.config.js.

webpack.dll.config.js

plugins: [

**new** webpack.DLLPlugin({

name: "[name]",

path: path.join(\_\_dirname, "[name]-manifest.json")

})

]

1. Eliminamos lineas de código de developmente en nuestro webpack.dll.config.js.
2. Corres el dll wepack config y despues corres el otro webpack config con el script que creaste para npm run en tu package.json.

package.json

"build:dll": "webpack --config ./dll/webpack.dll.config.js",

"build:prevent-duplication-fast": "webpack --config ./dll/webpack.config.js"

cli

$ npm run build:dll

$ npm run build:prevent-duplication-fast

1. Añades el siguiente plugin en tu archivo webpack.config.js. El plugin optimizará tu CSS para producción, no está directamente relacionado con el DLL.

**new** **MiniCssExtractPlugin**({

filename: "css/[name].css",

chunkFilename: "css/[id].css"

}) ,

Notas:

* library es quien enlaza los modulos que creamos al codigo que escribimos de la app
* Esta clase es dificil, practica, genera nuevas configuraciones.

No me gusta mucho el recurso de hardcodear los archivos html de nuestra carpeta raíz. Así que me propuse dar como resultado esos archivos en nuestro bundle y que todo quede en automático. Lo logré hacer así:  
.  
**1. Que como output se hagan dos archivos html dentro del dist: index.html y contact.html.** Esto se hace declarando el HtmlWebpackPlugin más de una vez.  
.  
**2. Que cada uno de esos archivos html mande a traer su propio js (index.js y contact.js).** Esto se hace declarando en la propiedad ‘chunks’ (de cada uno de los HtmlWebpackPlugin) el chunk correspondiente.  
.  
**3. Que los dos html resultantes manden a traer el modules.js de la carpeta dist en automático.** Esto se puede hacer con el plugin [AddAssetHtmlPlugin](https://github.com/SimenB/add-asset-html-webpack-plugin" \t "_blank). Se tiene que instalar a través de npm (las instrucciones vienen en el enlace).  
.  
Ésta es la parte modificada del código (webpack.config.js):

plugins: [

**new** MiniCSSExtractPlugin({

filename: 'css/[name].css',

chunkFilename: 'css/[id].css'

}),

**new** HtmlWebpackPlugin({

title: 'index',

template: path.resolve(\_\_dirname, 'index.html'),

chunks: ['home']

}),

**new** HtmlWebpackPlugin({

title: 'contact',

filename: 'contact.html',

template: path.resolve(\_\_dirname, 'contact.html'),

chunks: ['contact']

}),

**new** AddAssetHtmlPlugin({

filepath: require.resolve('./dist/js/modules.js'),

}),

**new** webpack.DllReferencePlugin({

manifest: require('./modules-manifest.json')

})

]

.  
Lo cual, da como resultado index.html dentro de dist:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Home</title>

<link href="css/home.css" rel="stylesheet"></head>

<body>

<div id="root"></div>

<script src="modules.js"></script><script src="js/home.js"></script></body>

</html>

.  
Y contact.html dentro de dist:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Contact</title>

<link href="css/contact.css" rel="stylesheet"></head>

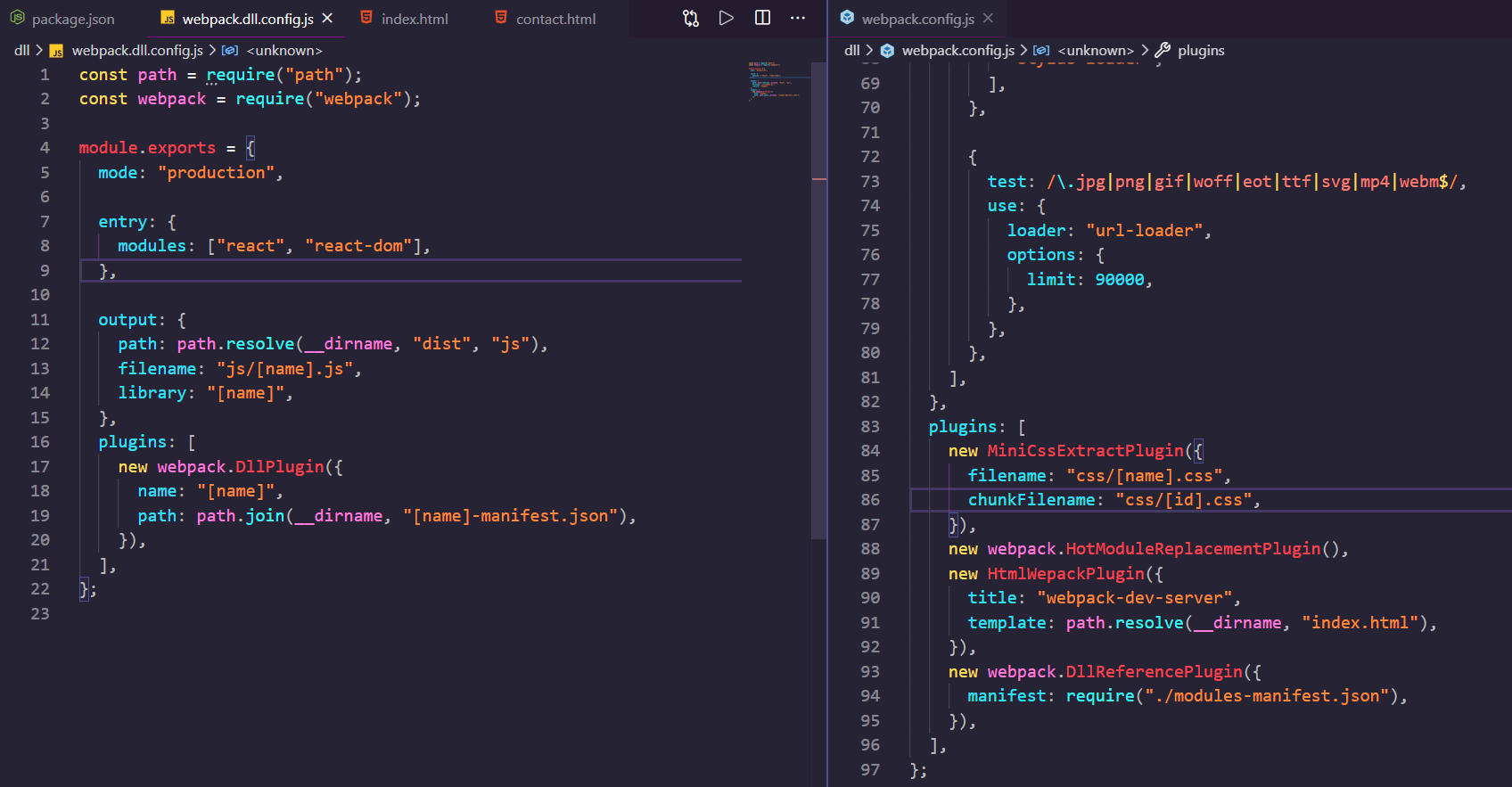
<body>

<div id="root"></div>

<script src="modules.js"></script><script src="js/contact.js"></script></body>

</html>

De esta forma, no es necesario estar hardcodeando los archivos html de la carpeta raíz, y tenemos lo que esperamos en la carpeta dist. Espero les ayude!



# Dynamic imports

Resumen de la clase

1)) Instalar **@babel/plugin-syntax-dynamic-import**

npm install --**save**-dev --**save**-exact @babel/**plugin**-**syntax**-dynamic-import

2)) En **.babelrc** agregar el plugin:

"plugins": [

"@babel/plugin-syntax-dynamic-import"

]

3)) En el atributo output de **webpack.config.js** agregar lo siguiente:

publicPath: 'dist/',

chunkFilename: 'js/[id].[chunkhash].js'

Pd: Si el error “*Module not found: Error: Can’t resolve '@babel/runtime/helpers/asyncToGenerator*’” ocurre, instalar **@babel/runtime**, así:

npm install --save-dev --save-exact @babel/runtime

