Apuntes del curso de Webpack de Platzi

**¿Qué es Webpack?**  
básicamente webpack es un **paquete** de módulos y esto lo que hace es que nuestra aplicación puede tener archivos JavaScript o jsx, archivos sass, imágenes y empaquetarlos como si fuera una caja (todo en uno)

Webpack es un module bundler. Los **module bundlers** son herramientas para frontend devs que suelen usarlo para empaquetar los *JS modules* en un único archivo de JS que pueda ser ejecutado por el navegador.

Las razones principales por que se usan webpack son:

* Algunos navegadores no soportan el sistema de modulos.
* Te ayudan a manejar las relaciones de dependencias con tu código.
* Ayudan a cargar assets en orden de dependencias, imágenes, css, etc.

Tener en cuenta que webpack es una filosofía que nos va permitir trabajar de forma modular. Es como menter todo lo que vamos a necesitar en un proyecto en una caja y después poder sacar todo listo para producción.

**Conceptos básicos**

**Entry** (punto de entrada): este le indica a webpack cual modulo de JavaScript debe de usar para empezar a crear una salida.  
**Ejemplo** : index.js. también podemos tener múltiples puntos de entrada pero eso es otra historia.  
.  
**Output** (punto de salida): Este archivo es el **bundle o mains.js** o nuestro archivo de salida, seria nuestra **caja** donde empaquetamos toda nuestra aplicación, normalmente este archivo final se crea en una carpeta llamada *dist*  
.  
**Loader** (transformador): Los loaders lo que hacen es decirle a webpack como tiene que transformar el código de un modulo en concreto. **Ejemplo** : Los loaders pueden transformar ficheros a JavaScript, o cargar **CSS** directamente en archivos JS, (si usas reactjs ya sabrás como)  
.  
**Plugins** (complementos): Nos van a ayudar a extender las funcionalidades con los loaders, añadir otras configuraciones.  
**Ejemplo** : hay un modulo llamado **HTMLWebpackPlugin** que este se encarga de crear un HTML personalizado que le inyecta todos los **bundles** finales que compilamos.

**Tu primer build con Webpack**  
Creamos una carpeta como le quieras llamar  
(Bueno no! si eres de Windows te dejo este articulo cortito de los nombres de carpetas [PROHIBIDOS](https://www.fayerwayer.com/2020/07/windows-nombres-prohibidos-en-carpetas/#:~:text=Nombres%20prohibidos%20para%20crear%20carpetas%20y%20archivos%20en%20Windows&text=CLOCK%24.,%2C%20LPT7%2C%20LPT8%2C%20LPT9.))  
La creamos desde la terminal con **mkdir** y luego entramos a ella con **cd**

mkdir curso-webpack

cd curso-webpack

una vez que entres a la carpeta inicializamos nuestro repositorio con **git**

git init

El paso que sigue es inicializar nuestro proyecto con npm y si no sabes de npm aqui esta el [curso](https://platzi.com/clases/npm/) del profesor

npm init -y

o si les da error “Invalid Name” usen para personalizar la configuración

npm init

y para abrir el proyecto **como flash** es poner en la terminal y les abre el editor ( si usas **VS CODE**)

**code** .

La carpeta **SRC** es el source de todo el proyecto ( index.js , imágenes, utils, assets, helpers, database, etc).

\*\* Instalación de Webpack\*\*  
si no quieres escribir ese comando también puedes usar este  
la **i** de install

npm i webpack webpack-**cli** -D

o si usas **yarn** usa

yarn **add** webpack webpack-**cli** -D

Y luego ejecutamos webpack  
**npx** lo que hace es ejecutar paquetes directamente de npm, este viene instalado de npm

npx webpack

Al hacer esto webpack creo una carpeta llamada **dist**, esto lo hace por defecto webpack sin preguntarnos.  
**Modo de desarrollo**  
Por defecto webpack al compilar nuestro proyecto setea el modo “production” implícitamente pero podemos definirle el modo explícitamente corriendo:

npx webpack --mode production

npx webpack --mode development

La **diferencia** radica que el modo development deja el código mas legible para los desarrolladores pero con comentarios, el modo **production** deja el código comprimido y mas limpio para usarse.

**Configuración de webpack.config.js**

* El flag —config indica donde estará nuestro archivo de configuración

**npx** **webpack** **--mode** **production** **--config** **webpack**.config.js

* Para poder hacerlo más amigable el comando puedes crear un script en package.json

"scripts": {

...

"build": "webpack --mode production --config webpack.config.js"

},

**💛 Babel Loader para JavaScript**

Es un transcompilador de JavaScript que agarra el código ECMAScript 2015 en adelante y lo transforma en una versión que todos los navegadores anteriores lo puedan usar

* Babel te permite hacer que tu código JavaScript sea compatible con todos los navegadores
* Debes agregar a tu proyecto las siguientes dependencias

NPM

npm install -D babel-loader @babel/core @babel/preset-env @babel/plugin-transform-runtime

Yarn

yarn add -D babel-loader @babel/core @babel/preset-env @babel/plugin-transform-runtime

* **babel-loader** nos permite usar babel con webpack
* **@babel/core** es babel en general
* **@babel/preset-env** trae y te permite usar las ultimas características de JavaScript
* **@babel/plugin-transform-runtime** te permite trabajar con todo el tema de asincronismo como ser async y await
* Debes crear el archivo de configuración de babel el cual tiene como nombre .babelrc

{

"presets": [

"@babel/preset-env"

],

"plugins": [

"@babel/plugin-transform-runtime"

]

}

* Para comenzar a utilizar webpack debemos agregar la siguiente configuración en webpack.config.js

{

...,

module: {

rules: [

{

// Test declara que extensión de archivos aplicara el loader

test: /\.js$/,

// Use es un arreglo u objeto donde dices que loader aplicaras

use: {

loader: "babel-loader"

},

// Exclude permite omitir archivos o carpetas especificas

exclude: /node\_modules/

}

]

}

}

**RESUMEN:** Babel te ayuda a transpilar el código JavaScript, a un resultado el cual todos los navegadores lo puedan entender y ejecutar. Trae “extensiones” o plugins las cuales nos permiten tener características más allá del JavaScript común

**MY NOTES FOR HTML IN WEBPACK**

Como vamos a preparar nuestro proyecto para que webpack pueda entenderlo y prepararlo

* Lo primero que hay que hacer es instalar el plugin que nos permite trabajar webpack con html

npm i html-webpack-plugin -D

* Después añadimos esta dependencia a la configuración de webpack

//traemos esta dependencia que la instalamos con un comando de npm previamente

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

plugins: [

//hacemos una instancia de lo que definimos en el inicio del archivo

// le anadimos por parametro un objeto donde vamos a tener las

//configuraciones que le vamos anadir a nuestro plugin

**new** HtmlWebpackPlugin({

//inyecta el bundle al template html

inject: **true**,

//la ruta al template html

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

//nombre final del archivo

filename: filename: './index.html'

})

]

Eliminamos el script de nuestro html porque webpack se encargara de insertar el script que compila en nuestro html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

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

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

<link rel="stylesheet" href="../src/styles/main.css">

<title>JS Portfolio</title>

</head>

<body>

<div id="main"></div>

<!-- <script type="module" src="../src//index.js"></script> -->

</body>

</html>

Compilamos webpack

npm run dev

Una vez compilamos nuestro archivo de webpack se insertara el main.js en nuetsro html

* agregamos otro script para ejecutar mas facil nuestros comando

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

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

"dev": "webpack --mode development"

},

**Loaders para CSS y preprocesadores de CSS**

Un **preprocesador** CSS es un programa que te permite generar CSS a partir de la syntax única del preprocesador. Existen varios preprocesadores CSS de los cuales escoger, sin embargo, la mayoría de preprocesadores CSS añadirán algunas características que no existen en CSS puro, como variable, mixins, selectores anidados, entre otros. Estas características hacen la estructura de CSS más legible y fácil de mantener.

**MY NOTES FOR CSS AND PREPROCETORS OF CSS 😄**

\*\*[Loaders](https://webpack.js.org/concepts/" \l "loaders" \t "_blank)\*\*Fuera de contexto, webpack ***solamente entiende JavaScript y JSON***. Los **loaders** nos permite procesar archivos de otros tipos para convertirnos en módulos válidos que serán consumidos por nuestras aplicaciones y agregadas como dependencias.

En alto nivel, los **loaders** poseen 2 configuraciones principales:

1. test - propiedad que identifica cuáles archivos deberán ser transformados
2. use - propiedad que identifica el *loader* que será usado para transformar a dichos archivos

\*\*[Plugins](https://webpack.js.org/concepts/" \l "plugins" \t "_blank)\*\*Mientras los *loaders* transforman ciertos tipos de módulos, los *plugins* **\_son utilizados para extender tareas específicas**, como la optimización de paquetes, la gestión de activos y la inyección de variables de entorno.

Una vez importado el plugin, podemos desear el personalizarlos a través de ***opciones***.

En esta clase vamos a aprender a añadir un loader para pode utilizar nuestro css y también un plugin para poder trabajar con css dividirlo en diferentes partes de nuestra aplicación y unirlo

**Es importante eliminar las conexiones de archivos a css o scripts de js porque webpack cuando se compile lo hace**

* Primero instalamos el loader

npm install mini-css-extract-plugin css-loader -D

* Eliminamos la conexion al css que esta en nuestro html porque webpack la generara automáticamente al compilarse
* En nuestro codigo en src/index.js importamos nuestros estilos

import Template from './templates/Template.js';

import './styles/main.css';

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

const main = null || document.getElementById('main');

main.innerHTML = await Template();

})();

* Vamos a añadir la configuración a webpack para que pueda identificar los elementos que acabamos de agregar.

Importamos la dependencia

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

* Agregamos el loader a nuestras reglas

{

test: /\.css$/i, //expresion regular

//cual es elemento que vamos a tenr

//Nosotros podemos usar el use con un objeto o un arreglo segun la configuracion

// del plugin

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

}

* Creamos en los plugins una instancia de nuestro dependencia que importamos

plugins: [

//hacemos una instancia de lo que definimos en el inicio del archivo

// le anadimos por parametro un objeto donde vamos a tener las

//configuraciones que le vamos anadir a nuestro plugin

**new** HtmlWebpackPlugin({

//inyecta el bundle al template html

inject: **true**,

//la ruta al template html

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

//nombre final del archivo

filename: './index.html'

}),

**new** MiniCssExtracPlugin(),

]

**Para stylus**

npm install stylus stylus-loader -D

Añadimos el loader al arreglo de loaders y modificamos un poco la expresion regular

{

test: /\.css|.styl$/i,

use: [MiniCssExtractPlugin.loader,

"css-loader",

"stylus-loader"]

},

**🗂️ Copia de archivos con Webpack**

* Si tienes la necesidad de mover un archivo o directorio a tu proyecto final podemos usar un plugin llamado “**copy-webpack-plugin**”
* Para instalarlo debemos ejecutar el comando para npm

npm i **copy**-webpack-**plugin** -D

* Para poder comenzar a usarlo debemos agregar estas configuraciones a webpack.config.js

...

const CopyPlugin = require('copy-webpack-plugin');

module.exports = {

...

plugins: [

**new** CopyPlugin({

patterns: [

{

**from**: path.resolve(\_\_dirname, "src", "assets/images"),

to: "assets/images"

}

]

}),

]

}

* Es importante las propiedades from y to
  + **From** ⇒ que recurso (archivo o directorio) deseamos copiar al directorio final
  + **To** ⇒ en que ruta dentro de la carpeta final terminara los recursos

**🎴 Loaders de imágenes**

* Puedes usar una forma de importar las imágenes haciendo un import de las mismas y generando una variable
* No es necesario instalar ninguna dependencia, webpack ya lo tiene incluido debemos agregar la siguiente configuración

module.exports = {

...

module: {

rules: [

{

test: /\.png/,

type: "asset/resource"

}

]

},

}

* Para empezar a usar esta configuración debemos importar la imagen de la siguiente forma

**import** github **from** '../assets/images/github.png';

* Para incluirlo en el HTML debes hacer lo siguiente

// Ejemplo en Vanilla JS

**const** imagen = `<img src=`${github}` />`;

// Ejemplo en React

<img src={`${github}`} />

**🔤 Loaders de fuentes**

* Cuando utilizamos fuentes externas una buena práctica es descargarlas a nuestro proyecto
  + Debido a que no hará un llamado a otros sitios
* Por ello es importante usarlo dentro de webpack
* Para esta tarea instalaras y usaras “file-loader” y “url-loader”

instalación con NPM

npm install url-loader file-loader -D

* Para aplicar esta configuración debes agregar la siguiente información

module.exports = {

entry: "./src/index.js",

output: {

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

filename: "main.js",

assetModuleFilename: 'assets/images/[hash][ext][query]'

},

module: {

rules: [

...

{

test: /\.(woff|woff2)$/,

use: {

loader: "url-loader",

options: {

// limit => limite de tamaño

limit: 10000,

// Mimetype => tipo de dato

mimetype: "application/font-woff",

// name => nombre de salida

name: "[name].[ext]",

// outputPath => donde se va a guardar en la carpeta final

outputPath: "./assets/fonts/",

publicPath: "./assets/fonts/",

esModule: **false**,

}

}

}

]

},

...

}

* Es importante que dentro de los estilos agregues @font-face

@font-face {

font-family: "Ubuntu";

src: url("../assets/fonts/ubuntu-regular.woff2") format('woff2'),

url("../assets/fonts/ubuntu-regular.woff") format('woff');

font-weight: 400;

font-style: normal;

}

**MY NOTES FOR OPTIMIZATION WITH WEBPACK 😄**

Por qué es importante usar Hashes en nuestros archivos?

* Los recursos que se guardan en memoria cache suceden cuando el navegador entra a un sitio por primera vez detecta los recursos y los guarda. Por ello la siguiente vez sera mucho más rápido porque estarán en memoria
  + La desventaja esta cuando sacamos una nueva versión, porque tendrán un mismo nombre evitando que se descargue los nuevos cambios, por lo tanto, el usuario no recibirá los nuevos cambios
  + Para que no haya conflictos con la cache una vez que tengamos nuestro proyecto en producción es importante darles un hash para cada nueva versión

Segun la documentación oficial de webpack nos comunica que actualmente terser-webpack-plugin viene incluido desde webpack 5 😄

Fuente:

[TerserWebpackPlugin](https://webpack.js.org/plugins/terser-webpack-plugin/)

Probando un poco más dentro del proyecto, para activar terser por defecto de webpack solo debemos usar el siguiente código

**module**.**exports** = {

...

optimization: {

minimize:**true**

}

}

Es decir que no podemos usar la propiedad minimizer: []. Pero si deseáramos personalizar la optimización y agregar plugins como ser css-minimizer-webpack-plugin ahi toca instalar y usar terser-webpack-plugin dentro de optimizations, otro caso de este tipo sería si desearas personalizar el plugin de terser

Una de las cosas mas importantes por la cual utilizamos webpack es la optimización de nuestro proyecto, en comprimir nuestro css, nuestro javascript y optimizar nuestras imagenes, entre otras caracteristicas.

Vamos a instalar unas dependencias que nos ayudaran con esto

npm install css-minimizer-webpack-plugin terser-webpack-plugin -D

* Ahora vamos a añadir nuestra configuración en nuestro archivo de webpack

//Importamos los plugins que acabamos de instalar

**const** CssMinimizerPlugin = require('css-minimizer-webpack-plugin');

**const** TerserPlugin = requiere('terser-webpack-plugin');

* Luego vamos a añadir esto a nuestro archivo como una nueva configuración llamada optimization

optimization:{

minimize: **true**,

minimizer: [

//Instanciamos las dependencias que estamos importando

**new** CssMinimizerPlugin(),

**new** TerserPlugin(),

]

}

Otra de las optimizaciones que podemos realizar para poder verificar las versiones que estamos trabajando tiene que ver mucho directamente con los hash

Asi que vamos a identificar cada bild de nuestro proyecto con un hash

* añadimos los hash en output

output:{

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

//cambios este elemento y le pondremos tanto el name para que lo identifique

//como la parte del contenthash para que nos muestre eso

filename: '[name].[contenthash].js',

//para insertar el cambio y mover las fuentes a otra carpte lo hacemos aqui

assetModuleFilename: 'assets/image/[hash][ext][query]'

},

* lo añadimos en nuestras fuentes

{

test: /\.(woff|woff2)$/,

use: {

loader: 'url-loader',

options:{

limit:10000,

mimetype:"application/font-woff",

name: "[name].[contenthash].[ext]",

outputPath: "./assets/fonts/",

publicPath: "./assets/fonts/",

esModule: **false**,

}

}

}

* Lo añadimos en la instancia de la dependencia que compila nuestro css

//Le anadimos una configuracion al plugin que nos permite compilar en css

**new** MiniCssExtracPlugin({

filename: 'assets/[name].[contenthash].css'

}),

Terminamos la configuracion y ahora compilamos webpack y podremos ver los archivos optimizados en dist/assets

**MY NOTES FOR WEBPACK ALIAS 😄**

En esta clase vamos a aprender a como utilizar alias para los paths que estamos utilizando

Para crear un alias debes agregar la siguiente configuración a webpack

resolve:{

extensions:['.js'],

//Agregamos una key alias a nuestro objeto resolve

//para ponerles nombres mas pequenos a las extensiones

//de nuestros archivos

alias:{

'@utils': path.resolve(\_\_dirname, 'src/utils/'),

'@templates': path.resolve(\_\_dirname, 'src/templates/'),

'@styles': path.resolve(\_\_dirname, 'src/styles/'),

'@images': path.resolve(\_\_dirname, 'src/assets/images/'),

}

}

Estos alias los vamos a poder utilizar dentro de nuestro proyecto para que cuando webpack lo prepare va a identificar que esta es la ruta a la que tenemos que entrar

* Luego agregamos estos alias en nuestras rutas en nuestro archivo de js

**import** Template **from** '@templates/Template.js';

**import** '@styles/main.css';

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

**const** main = **null** || document.getElementById('main');

main.innerHTML = **await** Template();

})();

* Lo agregamos en nuestro archivo que genera nuestro template

**import** getData **from** '@utils/getData.js';

//importamos la configuracion de las imagenes

**import** github **from** '@images/github.png';

**import** twitter **from** '@images/twitter.png';

**import** instagram **from** '@images/instagram.png';

Agregamons un punto para que nos agarre las fonts

publicPath: "../assets/fonts/",

**🔑 Variables de entorno**

* Es importante considerar las variables de entorno va a ser un espacio seguro donde podemos guardar datos sensibles
  + Por ejemplo, subir llaves al repositorio no es buena idea cuando tienes un proyecto open source
* Para instalar debemos correr el comando

NPM

npm install dotenv-webpack -D

* Posteriormente debemos crear un archivo .env donde estarán la clave para acceder a la misma y el valor que contendrán
* # Ejemplo
* API=https://randomuser.me/api/
  + Es buena idea tener un archivo de ejemplo donde, el mismo si se pueda subir al repositorio como muestra de que campos van a ir
* Una vez creado el archivo .env debemos agregar la siguiente configuración en webpack.config.js

...

const Dotenv = require('dotenv-webpack');

module.exports = {

...

plugins: [

**new** Dotenv()

],

}

* dotenv-webpack ⇒ Leera el archivo .env por defecto y lo agregar a nuestro proyecto
* Para usarlas debes hacer lo siguiente

**const** nombre = process.env.NOMBRE\_VARIABLE;

* Toda la configuración se podrá acceder desde process.env

**Webpack en modo desarrollo**

* Creamos un nuevo archivo:  
  **webpack.config.dev.js**
* Copiamos todo lo de **webpack.config.js** a el archivo que acabamos de crear.
* Borramos o comentamos el siguiente código, ya que no necesitamos optimizar para el modo de desarrollo (Queremos ver cuando funcionan las cosas).

optimization: {

minimize: **true**,

minimizer: [

**new** **CssMinimizerPlugin**(),

**new** **TerserPlugin**()

]

}

También borramos o comentamos por la misma razón:

**const** CssMinimizerPlugin = require('css-minimizer-webpack-plugin');

**const** TerserPlugin = require('terser-webpack-plugin')

* Seguido del atributo output añadimos:

output: {

},

**mode**: 'development',

* En package.json:

"dev": "webpack --config webpack.config.dev.js"

* y ejecutamos **npm run dev**