

# Webpack

## Server side Templating

Le client envoie une requête au serveur qui retourne le code html de la page. Toute la logique métier réside côté serveur et chaque écran correspond à un fichier sur le serveur.

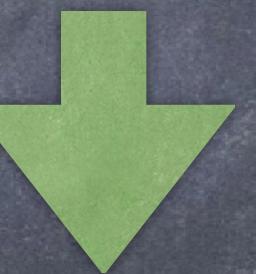
Javascript n'est utilisé que pour ajouter quelques éléments d'interactivité au sein de pages globalement statiques.



très peu de code javascript

## Single Page Application

Le client envoie une requête au serveur qui retourne un code html squelettique et un fichier javascript qui contient une bonne partie de la logique métier. Le serveur peut être à nouveau sollicité pour obtenir de nouvelles données, mais l'affichage et le traitement de celles-ci reste du ressort de Javascript



beaucoup de code javascript

## En production

On veut organiser l'application en une multitude de modules dans des fichiers séparés pour optimiser la **maintenance**

On veut utiliser des technos modernes (ES7, Sass, Less...) qui rendent le développement plus rapide et le code mieux organisé

On veut gérer les dépendances de nos divers fichiers tout en conservant notre santé mentale quand on passe en prod



productivité et organisation

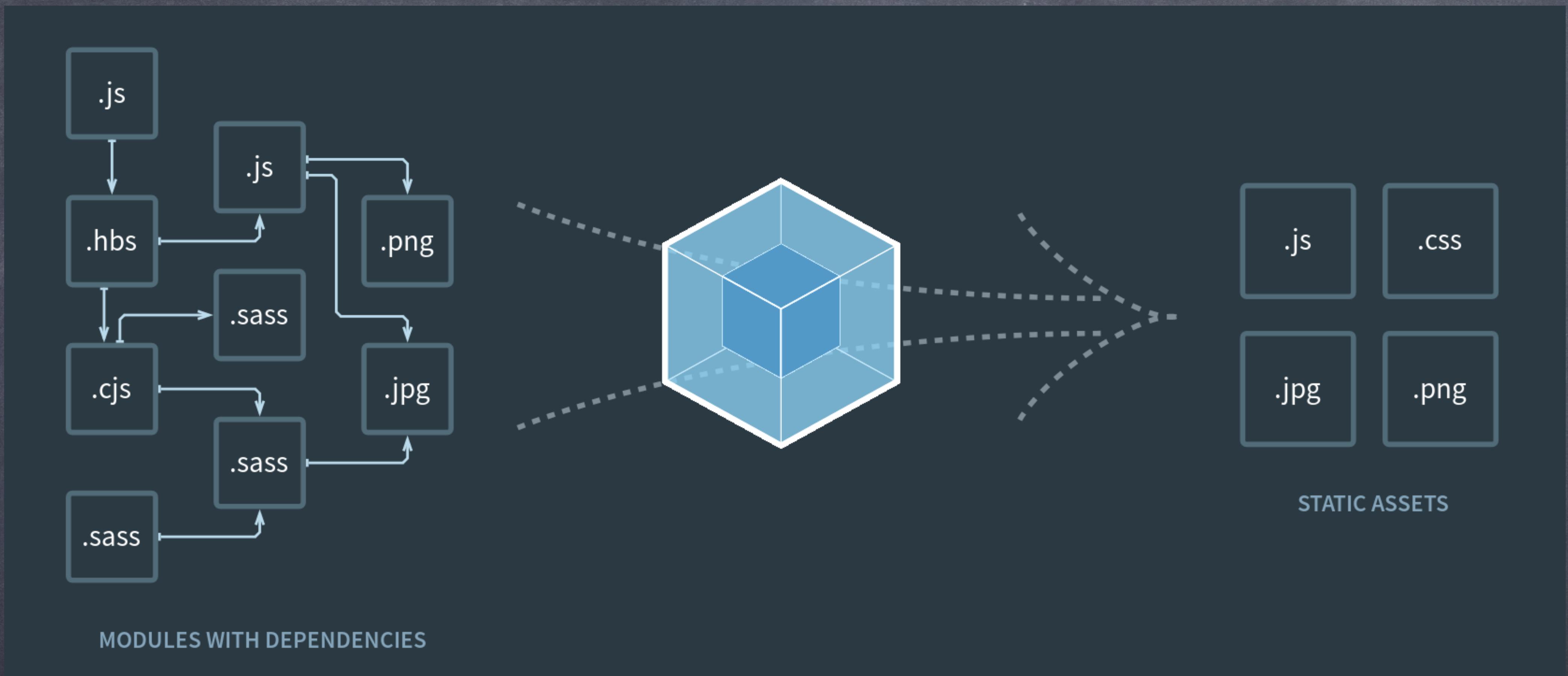
## En exploitation

On Veut minimiser le nombre de fichiers pour optimiser la **performance** en limitant les requêtes

On veut maximiser la compatibilité et la fiabilité de l'application



performance, compatibilité et fiabilité



Webpack est un bundler, un outil qui agrège de multiples fichiers en un seul gros fichier pour le déploiement d'une application web.

Webpack gère les dépendances des modules Javascript pour éviter les problèmes en production

Webpack peut également réaliser des traitements sur les fichiers avant de les agréger (minification, transpilation, ajout d'un hash...)

1. Créer un projet avec npm
2. Créer deux modules javascript
3. Installer et configurer Webpack
4. Lancer Webpack
5. Tester l'application

```
mkdir webpack-first-app  
cd webpack-first-app  
npm init -y
```

dans un terminal

Créer un dossier src dans le projet

```
export const sum = (...numbers) =>{
    return numbers.reduce((acc, n) => acc + n, 0);
};

export const product = (...numbers) => {
    return numbers.reduce((acc, n) => acc * n, 1);
};
```

src/calc.js

```
import {sum} from './app';

console.log('somme : ', sum(1,2,3));
```

src/app.js

```
npm install --save-dev webpack webpack-cli
```

dans un terminal

```
const path = require("node:path");
module.exports = {
  entry: './src/app.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  }
}
```

webpack.config.js

à la racine de l'application

# Webpack 4 Lancer Webpack

```
webpack --mode production | development
```

dans un terminal

Tester les modes production et dev,  
que constatez-vous dans le fichier bundle.js ?

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Webpack</title>
  <script type="module" src="./bundle.js"></script>
</head>
<body>
<h1>Test de webpack</h1>

</body>
</html>
```

index.html

dans le dossier dist,  
à tester avec un serveur web  
LiveServer par exemple

# Webpack Un serveur web pour les tests

Sébastien  
Maloron

12

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

dans un terminal

Déplacer le fichier index.html  
dans un dossier **public**  
à la racine de l'application

```
webpack-dev-server --mode production
```

dans un terminal pour lancer le serveur

# Webpack Pour servir index.html depuis le dossier dist

```
const path = require("node:path");
module.exports = {
  entry: './src/app.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  devServer: {
    static: path.resolve(__dirname, './dist'),
  }
}
```

webpack.config.js

à la racine de l'application

On peut remettre index.html  
dans le dossier dist  
et supprimer le dossier public

Le port utilisé par le serveur

ouvre le navigateur par défaut au lancement du serveur

```
devServer: {  
    static: path.resolve(__dirname, './dist'),  
    port: 8080,  
    open: true,  
    hot: true,  
    compress: true,  
}
```

active hot reload pour recharger les modules modifiés sans relancer le serveur

active la compression gzip

# Webpack Ajouter webpack dans les scripts

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "serve": "webpack-dev-server --mode development --watch",  
  "build": "webpack --mode production"  
},
```

package.json

```
npm run serve  
npm run build
```

dans un terminal

**Note**  
--watch pour relancer le build  
dès qu'un fichier est modifié

**Note**  
Pour arrêter le serveur CTRL+C

Par défaut Webpack ne traite que les fichiers Javascript et JSON et ne fait qu'agréger ces fichiers

Pour étendre les capacités de Webpack il faut utiliser un "loader"

# Webpack Ajouter un loader : exemple avec babel-loader

Sébastien  
Maloron

17

```
module: {  
  rules: [  
    {  
      'test': /\.js$/ ,  
      'exclude': /node_modules/ ,  
      use: {  
        loader: 'babel-loader' ,  
        options: {  
          targets: 'defaults' ,  
          presets: [  
            ['@babel/preset-env']  
          ]  
        }  
      }  
    }  
  ]  
}
```

Expression régulière indiquant à quels fichiers s'applique le loader

Expression régulière indiquant les dossiers et/ou fichier ignorés par le loader

Le nom du loader

The screenshot shows a web browser window displaying the npm package page for `babel-loader`. The URL in the address bar is `npmjs.com/package/babel-loader`, which is highlighted with a red box. At the bottom of the page, there is a red box highlighting the command `npm install -D babel-loader @babel/core @babel/preset-env webpack`.

**BABEL**

**Babel Loader**

This package allows transpiling JavaScript files using [Babel](#) and [webpack](#).

**Note:** Issues with the output should be reported on the Babel [Issues](#) tracker.

## Install

babel-loader	supported webpack versions	supported Babel versions	supported Node.js versions
8.x	4.x or 5.x	7.x	<code>&gt;= 8.9</code>
9.x	5.x	<code>^7.12.0</code>	<code>&gt;= 14.15.0</code>

```
npm install -D babel-loader @babel/core @babel/preset-env webpack
```

Repository [github.com/babel/babel-loader](https://github.com/babel/babel-loader)

Homepage [github.com/babel/babel-loader](https://github.com/babel/babel-loader)

Weekly Downloads 17934245

Version 9.2.1 License MIT

Unpacked Size 38.8 kB Total Files 10

Issues 59 Pull Requests 3

Last publish 14 days ago

Collaborators

<https://github.com/webpack/webpack>

>-Try on RunKit

```
npm install -D babel-loader @babel/core @babel/preset-env
```

dans un terminal

-D est un raccourci pour --save-dev

Refaire un build et constater la disparition  
du mot clef const et des arrow functions

```
body {  
    background: brown;  
}  
  
h1 {  
    color: orange;  
}
```

src/app.css

```
import './app.css';  
import { sum, product } from './calc';  
  
console.log('somme', sum(1,2,3));
```

src/app.js

A ce stade si on refait un build on obtient une erreur

```
npm install -D css-loader style-loader
```

traite les fichiers css

charge le css dans la page html

Copie le code CSS dans index.html

Ajouter un autre fichier CSS  
pour tester l'agrégation

```
{  
  test: /\.css$/,
  use: ['style-loader', 'css-loader']
}
```

webpack.config.js dans la clef rules

# Webpack Intégration d'un Préprocesseur CSS

Sébastien  
Maloron

23

```
$background: indigo;  
$text-color: orange;  
  
body {  
    background: $background;  
}  
  
h1 {  
    color: $text-color;  
}
```

src/app.scss

```
import './app.scss';  
import { sum, product } from './calc';  
  
console.log('somme', sum(1,2,3));
```

src/app.js

```
npm install -D sass-loader sass
```

Attention l'ordre est important, Webpack exécute les loaders de droite à gauche, ici il faut que sass-loader s'exécute avant css-loader

```
{  
  test: /\.scss$/ ,  
  use: ['style-loader', 'css-loader', sass-loader]  
}
```

webpack.config.js dans la clef rules

On a changé la config Webpack donc  
il faut relancer le serveur

- ✓ Pour Webpack une **ressource (asset)** est un fichier non exécutable (image, police de caractère ou simple fichier texte). Il y a 4 stratégies de gestion des assets
- ✓ **asset/ressource** : copie les fichiers dans le dossier dist
- ✓ **asset/inline** : copie le code de la ressource dans le fichier html (ok pour des petits fichiers, du svg par exemple)
- ✓ **asset** : choisit l'une des deux stratégies précédentes, si la ressource fait moins de 8 ko elle sera inline (on peut changer cette limite)
- ✓ **asset/source** : copie le contenu d'un fichier texte dans une variable Javascript

# Webpack Utilisation d'une image dans Javascript

```
import logo from './img/passiflore.jpg';

export const addImage = () => {
    const img = document.createElement('img');
    img.src = logo;
    const body = document.querySelector('body');
    body.appendChild(img);
}
```

Chercher une image sur le web

src/add-image.js

```
import './app.scss';
import { sum, product } from './calc';
import { addImage } from "./add-image";

console.log('somme', sum(1,2,3));

addImage();
```

src/app.js

# Webpack Utilisation d'une image dans Javascript

```
import logo from './img/passiflore.jpg';

export const addImage = () => {
  const img = document.createElement('img');
  img.src = logo;
  const body = document.querySelector('body');
  body.appendChild(img);
}
```

Chercher une image sur le web

src/add-image.js

```
import './app.scss';
import { sum, product } from './calc';
import { addImage } from "./add-image";

console.log('somme', sum(1,2,3));

addImage();
```

src/app.js

```
{  
  test: /\.png|jpg$/,
  type: 'asset/resource',
}
```

webpack.config.js dans la clef rules

Refaire un build et observer le  
contenu du dossier dist

Tester avec le type asset/inline  
l'image est sérialisée en base64 dans  
le fichier javascript

Quels sont les avantages et  
inconvénients de cette méthode ?

Télécharger la police Bangers depuis Google fonts et placer le fichier ttf dans src/fonts et faire un build

```
@font-face {  
    font-family: 'bangers';  
    src: url('fonts/Bangers-regular.ttf');  
}  
  
$background: indigo;  
$text-color: orange;  
  
body {  
    background: $background;  
}  
  
h1 {  
    color: $text-color;  
    font-family: bangers;  
}
```

app.scss

```
{  
    test: /\.(png|jpg|ttf)$/,  
    type: 'asset',  
}
```

webpack.config.js

La police fait plus de 8Ko donc c'est asset/resource qui est utilisé, on peut changer cette limite

```
{  
  test: /\.png|jpg|ttf$/,
  type: 'asset',
  parser: {
    dataUrlCondition: {
      maxSize: 100 * 1024, // 100 Ko
    }
  }
}
```

webpack.config.js dans la clef rules

La police fait 92 Ko  
elle sera donc inline  
passiflore.jpg fait plus  
elle ne sera donc pas inline

Hello webpack

src/message.txt

```
{  
  test: /\.txt$/,
  type: 'asset/source',
}
```

webpack.config.js dans la clef rules

```
import './app.scss';
import { sum, product } from './calc';
import { addImage } from "./add-image";
import message from "./message.txt";

console.log('somme', sum(1,2,3));
addImage();

document.querySelector('h1').innerHTML = message;
```

src/app.js

Pour faire tout ce que les loaders ne savent pas faire

- ✓ Les loaders travaillent sur les fichiers individuels et les agrègent en paquets (bundle)
- ✓ Les plugins interviennent sur les paquets générés pour les modifier
- ✓ Les plugins étendent les possibilités de Webpack

- ✓ Installer la dépendance avec npm install
- ✓ Dans webpack.config.js
  - ✓ importer la dépendance avec require
  - ✓ instancier le plugin dans la clef plugin de la configuration

Supprime les espaces et  
renomme les variables pour  
réduire la taille du fichier

```
npm install -D terser-webpack-plugin
```

Téléchargement

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  plugins: [
    new TerserPlugin(),
  ]
}
```

Importation

Instanciation

Attention, ce fichier est incomplet

Terser minified le code Javascript, pour le débogage c'est assez gênant

Source-map permet de récupérer des infos de debug plus pertinentes dans notre navigateur

```
module.exports = {  
  devtool: 'source-map'  
}
```

Extrait le contenu CSS dans un fichier bundle au lieu de l'inclure dans le code javascript

npm install -D mini-css-extract-plugin

Téléchargement

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader']
      },
      {
        test: /\.scss$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader', 'sass-loader']
      }
    ],
    plugins: [
      new MiniCssExtractPlugin({
        filename: 'bundle.css'
      })
    ]
  }
}
```

Importation

Utilisation dans les loaders

Instanciation

Il faut modifier le document html pour lier la nouvelle feuille de style

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Webpack</title>
  <script type="module" src=".bundle.js"></script>
  <link rel="stylesheet" href=".bundle.css">
</head>
<body>
<h1>Test de webpack</h1>

</body>
</html>
```

Essayez d'ajouter un nouveau fichier CSS au projet pour tester l'agrégation en un seul bundle

Modifier la limite de taille du loader des asset pour extraire la police du bundle CSS

Les navigateurs stockent les ressources (js, css, images ...) dans leur cache pour limiter la consommation de bande passante et améliorer la performance des application

Problème, quand on corrige un bug ou ajoute une fonctionnalité il faut forcer le navigateur à charger la nouvelle version et pas celle en cache

La solution la plus fréquente est d'ajouter un hash au fichier, ce hash dépendant du contenu du fichier, si celui-ci ne change pas le nom reste identique

```
entry: './src/app.js',  
output: {  
  path: path.resolve(__dirname, 'dist'),  
  filename: 'bundle-[contenthash].js',  
}
```

- Faire un build et constater le résultat
- Refaire le build et constater que le nom ne change pas
- Modifier app.js et refaire un build, cette fois ci le nom change

Faites la même chose pour le bundle CSS

Supprime les anciens fichiers avant la création des nouveaux bundles

```
npm install -D clean-webpack-plugin
```

```
const { CleanWebpackPlugin } = require('clean-webpack-plugin');
module.exports = {
  plugins: [
    new CleanWebpackPlugin()
  ]
}
```

Notez les accolades  
dans l'import

```
entry: './src/app.js',
output: {
  path: path.resolve(__dirname, 'dist'),
  filename: 'bundle-[contenthash].js',
  clean: true
}
```

On a perdu notre fichier index.html que l'on avait copié dans dist

```
npm install -D html-webpack-plugin
```

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  plugins: [
    new HtmlWebpackPlugin()
  ]
}
```

Refaire un build et constater que nous avons un fichier index.html avec les liens vers nos bundles CSS et JS

Copier index.html dans le dossier src  
Supprimer les balises link et script  
Ajouter un h1

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Webpack first app</title>
</head>
<body>
<h1>Test de webpack</h1>

</body>
</html>
```

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ]
}
```

Refaire un build et constater que nous avons un fichier index.html avec les liens vers nos bundles CSS et JS

```
npm install -D copy-webpack-plugin
```

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

new CopyPlugin({
  patterns: [
    {from: './src/assets', to: './assets'}
  ]
})
```

Créer un dossier assets et un sous dossier img dans src

Dans ce dossier img placer une image

Ajouter cette image à index.html

Faire un build et constater le résultat

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

new CopyPlugin({
  patterns: [
    {
      from: "./src/assets",
      to: "./assets/[path][name].[contenthash].[ext]",
    }
  ]
})
```

Découpage des sources  
en plusieurs fichiers

- ✓ Certains modules changent moins souvent (les vendors par exemple)
- ✓ Certains modules pourraient être chargés à la demande (lazy loading)
- ✓ Certains modules sont partagés par plusieurs pages

# Webpack Exemple

On sépare les outputs avec un nom et un hash

Le tableau chunks référence les scripts qui doivent être insérés dans le fichier html

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

module.exports = {
  entry: {
    app: './src/index.js',
    quill: 'quill',
  },
  output: {
    filename: '[name].[contenthash].js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  optimization: {
    splitChunks: {
      chunks: 'all'
    },
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
      chunks: ['app', 'quill'],
    }),
  ],
  module: {
    rules: [
      { test: /\.css$/, use: ['style-loader', 'css-loader'] },
    ],
  },
  mode: "development"
};
```

Autant d'entrées que de bibliothèques

Activation du découpage en chunks

lien vers le projet

lien vers le projet

Quel est l'intérêt par rapport à la solution précédente ?

Le tableau chunks référence le fichier de l'application et l'agrégation des dépendances

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

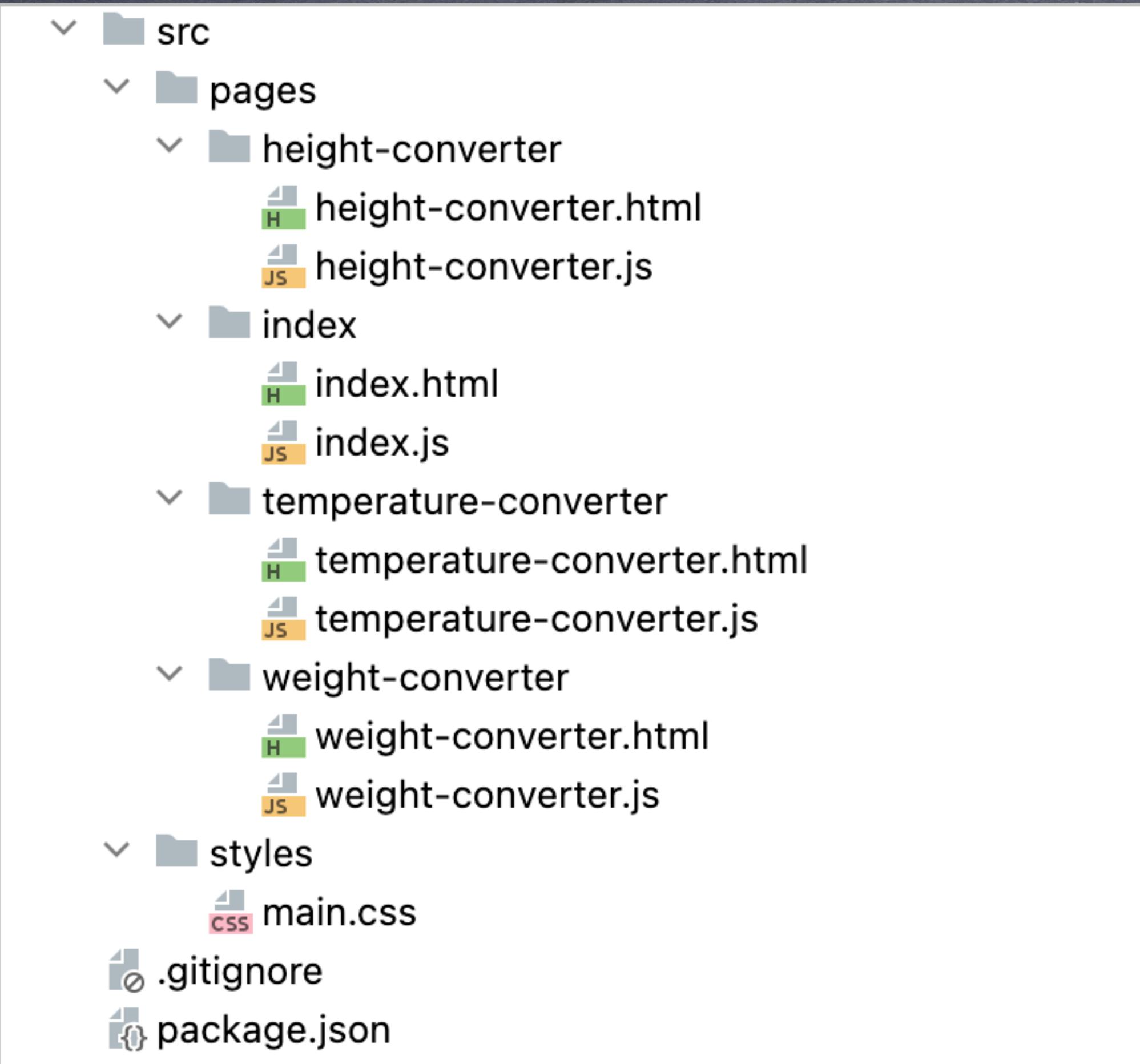
module.exports = {
  entry: {
    app: './src/index.js',
  },
  output: {
    filename: '[name].[contenthash].js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  optimization: {
    splitChunks: {
      chunks: 'all',
      cacheGroups: {
        vendors: {
          test: /[\\/]node_modules[\\/]/,
          name: 'vendors',
          chunks: 'all'
        }
      }
    }
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
      chunks: ['app', 'vendors'],
    }),
  ],
  mode: "development"
};
```

Ici une seule entrée pour le code de l'application

Avec cacheGroups on agrège l'ensemble des bibliothèques contenues dans nodes\_modules en un seul fichier

# Pages multiples

<https://github.com/smoloron/webpack-multi-page/tree/master>



npm install

- ✓ Dans la clef entry, on passe un objet avec en clef le nom du fichier et en valeur son chemin relatif
- ✓ Dans les plugins on référence une instance de HtmlWebpackPlugin par page html

Essayez par vous même,  
penser aux chunks  
pour extraire  
les fichiers js séparés

# Webpack Une solution

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  entry: {
    index: './src/pages/index/index.js',
    'height-converter': './src/pages/height-converter/height-converter.js',
    'temperature-converter': './src/pages/temperature-converter/temperature-converter.js',
    'weight-converter': './src/pages/weight-converter/weight-converter.js'
  },
  output: {
    filename: '[name].[contenthash].js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  module: {
    rules: [
      {test: /\.(png|jpg|ttf)$/, type: 'asset'},
      {test: /\.css$/, use: [MiniCssExtractPlugin.loader, 'css-loader']},
    ]
  },
  optimization: {
    splitChunks: {
      chunks: 'all',
    }
  },
}
```

```
plugins: [
  new MiniCssExtractPlugin({
    filename: 'bundle-[contenthash].css'
  }),
  new HtmlWebpackPlugin({
    template: './src/pages/index/index.html',
    filename: 'index.html',
    chunks: ['index'],
  }),
  new HtmlWebpackPlugin({
    template: './src/pages/height-converter/height-converter.html',
    filename: 'height-converter.html',
    chunks: ['height-converter'],
  }),
  new HtmlWebpackPlugin({
    template: './src/pages/weight-converter/weight-converter.html',
    filename: 'weight-converter.html',
    chunks: ['weight-converter'],
  }),
  new HtmlWebpackPlugin({
    template: './src/pages/temperature-converter/temperature-converter.html',
    filename: 'temperature-converter.html',
    chunks: ['temperature-converter'],
  })
],
```

[lien vers la solution](#)

lien vers la solution