



**webpack**



# Czym jest webpack? Po pierwsze to module bundler

Webpack to **module bundler**, po polsku możemy powiedzieć “pakowacz modułów”. Pozwala on wiele modułów (w tym również zewnętrznych paczek) JavaScript skompilować, **uwzględniając zależności**, do jednego lub wielu plików (pakietów).

Słowniczek:

**bundling** - łączenie, kompletowanie, pakowanie

**bundler** - pakowacz

**bundle (bundles)** - pakiet (pakiety)

**bundle modules** - łączyć moduły

**assets** - zasoby

“zbundlować”, “bundlować” - spakować, pakować



<https://webpack.js.org>

# Czym jest webpack? Po drugie automatyzacja zadań

Webpack pomaga też automatyzować zadania, czyli pozwala podczas bundlowania, także z użyciem zewnętrznych narzędzi, m.in. na wykonywanie kompilacji sass na css, transpilacji ES6+ na ES5. Webpack zoptymalizuje też wersję produkcyjnej aplikacji (m.in. minifikacja, uglifikacja, rozmiar).

Webpack umożliwia również uruchomienie web serwera developerskiego. Pozwala też korzystać z narzędzi przydatnych przy tworzeniu aplikacji jak linter.



Sass



BABEL

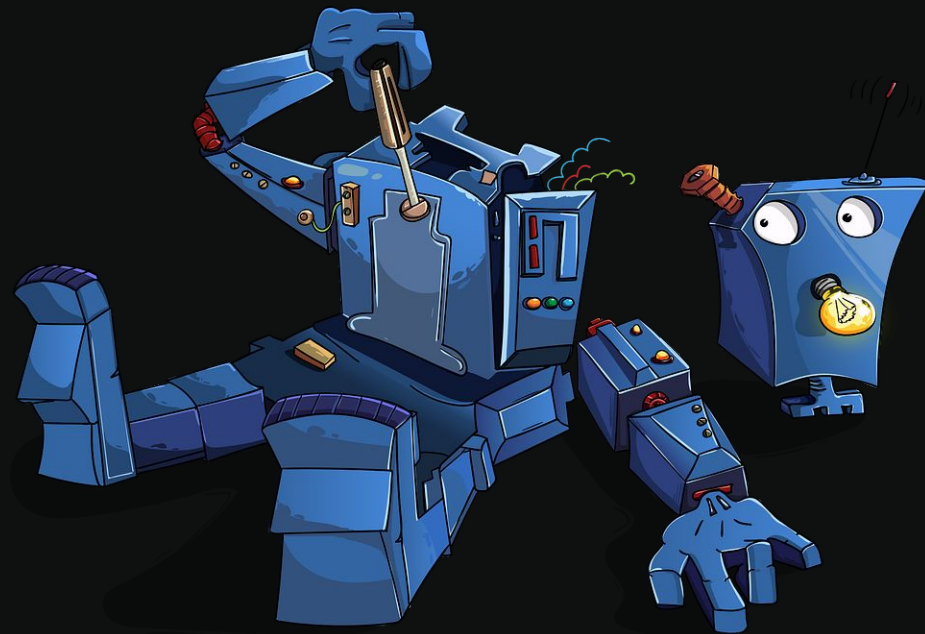
# W czym pomoże nam webpack

- Kod aplikacji będzie prawidłowo odczytany przez przeglądarkę.
- Kod aplikacji będzie zoptymalizowany.
- Wiele zadań, które musimy zrobić na etapie developmentu może być dzięki webpackowi (i dodatkowym pluginom) zautomatyzowana - i tutaj webpack spełnia rolę, którą spełniają tzw. **task runnery**.

# Czyli co robi ten webpack??

webpack pozwala Ci **pisać i wykorzystywać dostępne moduły**, które potem (po bundlingu) **zadziałają w przeglądarce**. A przy okazji zoptymalizuje wszystko.

Traktuj webpack jako **narzędzie**, które pomaga Ci zapanować zarówno nad złożonością procesu tworzenia strony/aplikacji jak również licznymi technologiami, których używasz. Oczywiście zanim webpack zacznie nam pomagać musimy go ujarzmić. Webpack tworzy dodatkową warstwę abstrakcji w postaci **konfiguracji**, co jest pewną barierą wejścia dla osób, które zaczynają z nim pracować. Jednak warto się przez nią przebić.



# ES Modules i CommonJS

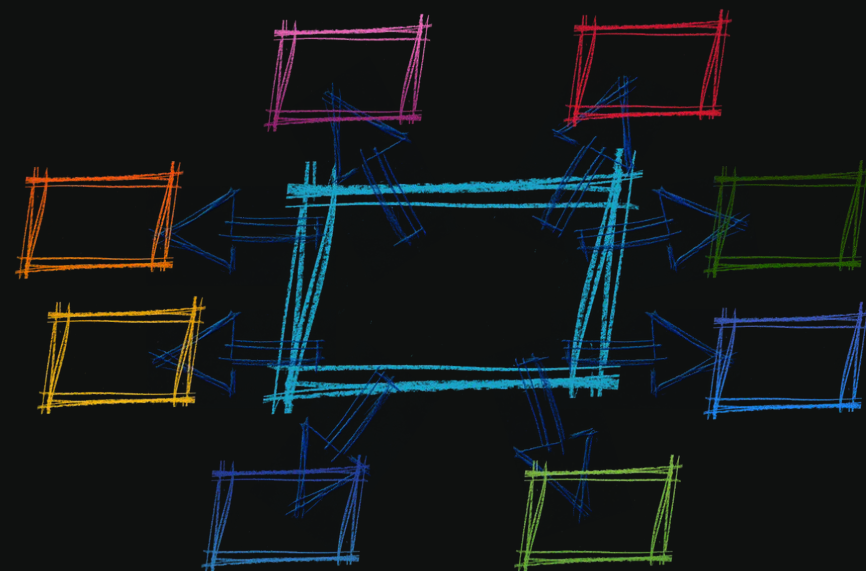
- ES Modules są rekomendowane do użycia wraz z webpackiem, który sobie od dawna dobrze z nimi radzi. Zamiast funkcji `require` i `module.exports` (CommonJS) w naszych modułach, o ile korzystamy z webpacka, będziemy najczęściej używać instrukcji `import` i `export` - poznaliśmy składnię ESM w poprzedniej sekcji.
- Używanie innych specyfikacji modułów (jak CommonJS czy AMD) jest oczywiście poprawne i webpack je rozpozna. Pamiętajmy, że paczki z npm których korzystamy są obecnie zbudowane przede wszystkim na specyfikacji CommonJS.
- webpack potrafi sobie świetnie radzić gdy w jednym projekcie występują różne specyfikacje modułów (ESM, CommonJS, AMD).

# Zależności i wykres zależności (dependency graph)

Webpack traktuje (Twój) projekt jako wykres zależności - **dependency graph**, to nazwa, którą wypada zapamiętać.

Na wejściu (początek bundlingu) znajduje się jakiś plik (entry point) javascriptowy. Ona ma liczne zależności (coś importuje) - przede wszystkim inne moduły JavaScript, ale także m.in. grafiki, pliki css/sass czy biblioteki/frameworki, których używamy. Oczywiście kolejne zależności znajdują się też w innych modułach.

W procesie kompilacji webpack tworzy pełen **wykres zależności**, który przypomina trochę drzewo, trochę sieć. Webpack wszystkie te zależności potrafi rozpoznać. Dzięki temu “rozumie” działanie aplikacji i potrafi zrobić prawidłowy i zoptymalizowany bundling. To jest pierwsza, podstawowa, fundamentalna cecha webpacka.





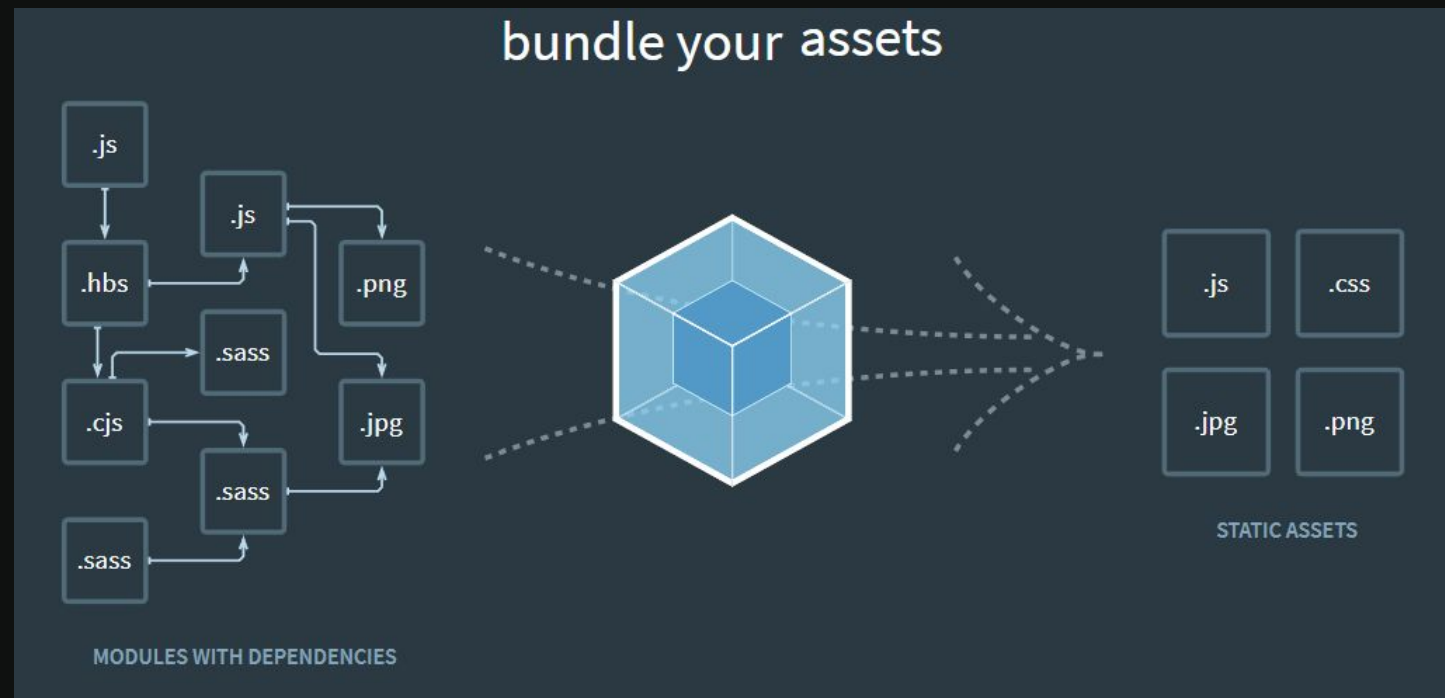
# Jak działa webpack

- webpack pracę zaczyna od pliku wejściowego (tych plików może być wiele)
- webpack parsuje pliki i buduje wykres zależności (akceptuje ES Module czy CommonJS oraz mniej popularną specyfikację AMD). Webpack weźmie pod uwagę tylko moduły, które rzeczywiście są używane i znajdują się w drzewie zależności. Jeśli coś jest zainstalowane, czy zaimportowane ale nie jest używane to webpack może tego nie bundlować (co zależy od konfiguracji).
- webpack pakuje wszystkie potrzebne zależności JavaScriptowe do jednego pliku (lub wielu, jeśli tak zdecydujemy). Webpack łączy moduły we właściwej kolejności. Przy okazji procesu pakowania webpack może zminimalizować i zoptymalizować kod oraz wykonać na nim inne działania (transpilacja, kompilacja).



# JavaScript, ale nie tylko

Webpack domyślnie obsługuje JavaScript (i JSON), ale inne zasoby jak grafiki czy pliki css też mogą być obsługiwane - wymaga to użycia tzw. loaderów.





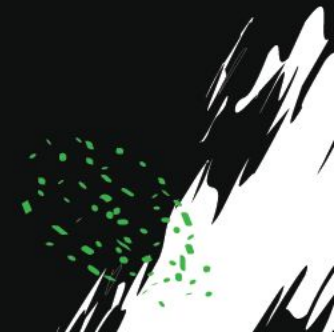
# Instalacja lokalna webpacka

rozpoczynamy pracę z projektem tworząc package.json

**npm init**

instalujemy lokalnie dwie paczki

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



# npm i webpack -D - czym jest ta paczka

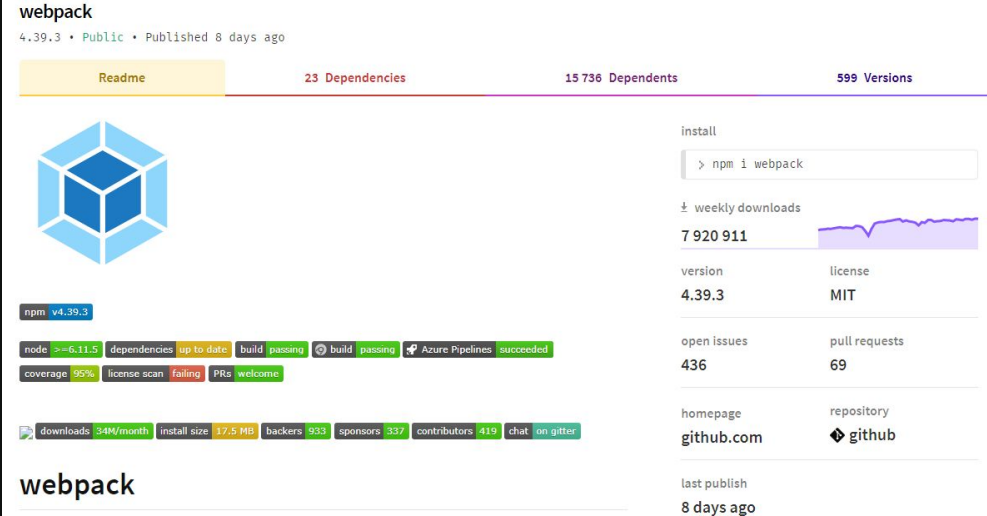
*Cytat ze strony paczki webpack w repozytorium npm:*

Bundles ES Modules, CommonJS, and AMD modules (even combined).

Can create a single bundle or multiple chunks.

Loaders can preprocess files while compiling, e.g. TypeScript to JavaScript, etc.

Highly modular plugin system to do whatever else your application requires.



The screenshot shows the npm page for the 'webpack' package. At the top, it displays the package name 'webpack', version '4.39.3', and status 'Public'. Below this, it shows '23 Dependencies', '15 736 Dependents', and '599 Versions'. The main content area features the webpack logo, a 'Readme' tab, and a list of badges for npm, node, dependencies, build, coverage, license scan, PRs, downloads, install size, backers, sponsors, contributors, chat, and gitter. On the right side, there is an 'install' section with a command prompt showing '> npm i webpack', a 'weekly downloads' graph showing 7 920 911 downloads, and a table with version 4.39.3, license MIT, open issues 436, pull requests 69, homepage github.com, repository github, and last publish 8 days ago.

version	license
4.39.3	MIT

open issues	pull requests
436	69

homepage	repository
github.com	github

last publish: 8 days ago

<https://www.npmjs.com/package/webpack>


# npm i webpack-cli -D - czym jest ta paczka

*Cytat ze strony paczki webpack-cli w repozytorium npm:*

webpack CLI provides a flexible set of commands for developers to increase speed when setting up a custom webpack project.

*Cytat ze strony webpack.js.org*

If you're using webpack v4 or later, you'll also need to install the CLI.



The screenshot shows the npm package page for webpack-cli. At the top, it displays the package name 'webpack-cli', version '3.3.7', and status 'Public'. Below this, there are statistics: '11 Dependencies', '2007 Dependents', and '84 Versions'. The main section features the webpack logo and the text 'webpack CLI' and 'The official CLI of webpack'. On the right, there is an 'Install' section with a command box containing '> npm i webpack-cli'. Below this is a 'weekly downloads' graph showing a peak of '2 433 550'. Further down, there are statistics for 'version 3.3.7', 'license MIT', 'open issues 62', and 'pull requests 9'. At the bottom, there are badges for 'npm v3.3.7', 'build passing', 'Azure Pipelines succeeded', 'dependencies', 'maintainability', 'chat', 'install size 3.96 MB', 'downloads 2.4M/week', 'maintained with karma', and 'contributors 83'. The footer includes 'homepage github.com' and 'repository github'.

npm	v3.3.7	build	passing	Azure Pipelines	succeeded	dependencies	repo or path not found or david internal error
maintainability	D	chat	on github	install size	3.96 MB	downloads	2.4M/week
maintained with	karma	contributors	83				

<https://www.npmjs.com/package/webpack-cli>

## Dodawanie paczek do projektu - wybór między statusem dependencies i devDependencies (czyli sposób zapisania w package.json)

Jeśli instalujemy paczkę, która będzie bundlowana do wersji produkcyjnej naszego projektu, to powinniśmy ją zainstalować do dependency. Jeśli paczka będzie używana tylko do celów developmentu, to zamieszczamy ją w **devDependency** (z opcją `--save-dev` lub `-D`). Konfigurując webpack (instalując pluginy i loadery - czym są, dowiesz się wkrótce) będziemy więc korzystać z **devDependencies** a dodając różne biblioteki (jak jQuery czy normalize.css) umieścisz je w **dependencies**.

```
"devDependencies": {  
  "@babel/core": "^7.5.5",  
  "@babel/preset-env": "^7.5.5",  
  "babel-loader": "^8.0.6",  
  "clean-webpack-plugin": "^3.0.0",  
  "css-loader": "^3.2.0",  
  "file-loader": "^4.2.0",  
  "html-webpack-plugin": "^3.2.0",  
  "style-loader": "^1.0.0",  
  "webpack": "^4.39.3",  
  "webpack-cli": "^3.3.7",  
  "webpack-dev-server": "^3.8.0"  
}
```

# webpack zadziała bez żadnej konfiguracji (kodu)?

Przejdźmy do VSC i sprawdźmy

stwórzmy index.html i app.js i tools.js - połączmy

uruchomimy webpacka za pomocą skryptu z package.json

przeanalizujemy błędy i zrobmy tak, by wszystko działało dobrze



Różne ustawienia w webpacku

Ale ciągle bez pliku konfiguracyjnego





# Różne ustawienia w webpacku za pomocą CLI

scripts

```
"test1": "webpack ./src/index.js -o ./distribution/bundle.js",
```

```
"test2": "webpack src/index.js --output dist/bundle.js",
```

```
"test3": "webpack --output-path bundles",
```

```
"test4": "webpack --output-path ./bundles",
```

```
"test5": "webpack --output-path bundles --output-filename script.js",
```

```
"test6": "webpack --output bundle/index.js --progress",
```

```
"test7": "webpack -o bundle/index.js",
```

```
"test8": "webpack --entry ./src/index.js --output bundless/index.js --mode development",
```

```
"test9": "webpack --mode=production --entry=./src/index.js --output=bundle/index.js"
```

# webpack 4 nie potrzebuje pliku konfiguracyjnego

Dlaczego webpack nie potrzebuje pliku konfiguracyjnego, ani nawet podawania parametrów w CLI? Ponieważ ma domyślnie ustawione wiele właściwości, między innymi co jest plikiem wejściowym (entry) i gdzie ma umieścić pliki po bundlingu (output).

**entry point:** ./src/index.js

**output path:** ./dist - dist to skrót od distribution

**output file:** main.js

W praktyce niemal zawsze będziesz taki plik konfiguracyjny tworzyć czy po prostu używać pliku, który stworzyłeś wcześniej (czy którego używa się w danym projekcie).

# webpack bootstrap / webpack runtime

Częścią kodu wyjściowego jest skrypt startowy webpacka - nawet jeśli input zawiera jedynie pusty moduł to output będzie zawierał taki kod startowy webpacka.

Dzięki temu “opakowaniu” możliwe jest uruchomienie kodu w przeglądarce.

# Uruchomienie webpacka - 3 sposoby

# Nowy skrypt w package.json i uruchomienie w konsoli

```
{  
  "name": "bundler",  
  "version": "1.0.0",  
  "main": "index.js",  
  "scripts": {  
    "build": "webpack"  
  },  
  "devDependencies": {  
    "webpack": "^4.39.2",  
    "webpack-cli": "^3.3.7"  
  }  
}
```

```
PS D:\projekty\kursy\css-js\webpack\bundler> npm run build
```

# npx webpack

Możemy użyć webpacka bez dodawania skryptu. Wystarczy wpisać polecenie npx webpack.

```
PS D:\projekty\kursy\css-js\webpack\bundler> npx webpack
```

to też zadziała:

```
npx webpack --entry ./src/index.js --output bundless/index.js --mode development
```

# bezpośrednie odwołanie

Możemy użyć wpisać też polecenia będącego adresem do webpacka

```
D:\projekty\kursy\css-js\webpack\050922019> node_modules/.bin/webpack
```

```
np. node_modules/.bin/webpack src/index.js --o dist/index.js
```

Przy czym to rozwiązanie nie zawsze musi zadziałać. Może pojawić się taki błąd (w PowerShellu):

```
node_modules/.bin/webpack : File /file name/ cannot be loaded because running scripts is disabled on this system.
```

W takim wypadku możesz uruchomić (jako administrator) w wierszu poleceń PowerShella, polecenie:

```
Set-ExecutionPolicy RemoteSigned
```



# Stwórzmy plik konfiguracyjny - zrobmy bundling!

Po uruchomieniu webpack szuka pliku konfiguracyjnego (domyślnie pod adresem `webpack.config.js`). Wiemy już, że jeśli go tam nie znajdzie (lub w innym wskazanym miejscu), zastosuje ustawienia domyślne. Jeśli chcemy pracować z webpackiem w bardziej zaawansowany sposób, plik konfiguracyjny jest niezbędny.

# Konfiguracja skryptów

# ustawmy w skryptach mode na różne wartości

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

W wierszu poleceń możemy w takim przypadku wpisać:

```
npm run build
```

```
npm run dev
```

Istnieją trzy wartości mode: production, development i none (wartość oznaczająca brak optymalizacji dla żadnego z dwóch trybów).

Zapis ze znakiem = jest też poprawny, czy to w skryptach czy bezpośrednio w wierszu poleceń np.

```
"dev": "webpack --mode=development"
```

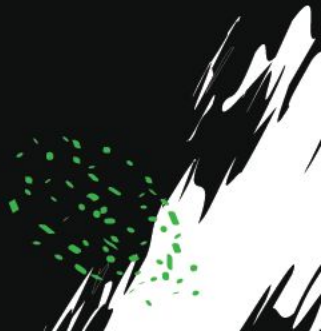
# Użycie skryptów dla różnych plików konfiguracyjnych

```
// package.json - przykładowe skrypty
```

```
"scripts": {  
  "watch": "webpack --watch",  
  "build": "webpack",  
  "start": "webpack-dev-server --mode development --devtool inline-source-map --open ",  
  "dev:prod": "webpack-dev-server --mode=production --open ",  
  "dev-server": "webpack-dev-server",  
  "transpile": "babel"  
}
```



Plik konfiguracyjny



# kilka plików konfiguracyjnych

Przykładowo możemy stworzyć dwa pliki konfiguracyjne. Dla wersji developerskiej (np. webpack.config.js) i dla wersji produkcyjnej (np. webpack.config.prod.js). Możemy potem dodać jest w skrypcie (package.json)

```
"scripts": {  
  "dev": "webpack",  
  "prod": "webpack --config webpack.config.prod.js",  
}
```

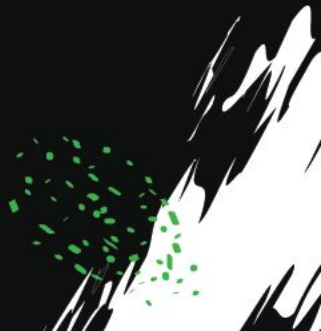
```
//domyślny plik konfiguracyjny to webpack.config.js, ale możemy to  
zmienić za pomocą opcji --config i wskazania nowej nazwy (ścieżki)  
do pliku.
```

```
//webpack.config.prod.js  
module.exports = {  
  mode: 'production',  
}
```

```
//webpack.config.js  
module.exports = {  
  mode: 'development',  
  devtool: inline-source-map,  
}
```



Użycie skryptów dla różnych plików konfiguracyjnych





# Użycie skryptów dla różnych plików konfiguracyjnych

Dzięki skryptom w package.json różni deweloperzy mogą pracować na tych samych zestawach poleceń

npm run - lista wszystkich skryptów

npm run build - wykonanie wskazanego skryptu

```
PS D:\projekty\kursy\css-js\webpack\app> npm run
Scripts available in webpack-app via `npm run-script`:
dev
  webpack --mode development
prod
  webpack --mode production
devtool
  webpack --mode development --devtools false
server
  webpack-dev-server --mode development
compile-and-run
  webpack --mode development && webpack-dev-server --mode development
```

# Najważniejsze koncepcje webpacka

# entry (point/points)

Webpack musi wiedzieć który plik JavaScript (moduł) jest plikiem wejściowym. Od niego zaczyna tworzenie grafu (wykresu) zależności.

Domyślnie tego pierwszego pliku webpack szuka pod adresem `./src/index.js` (w katalogu, w którym jest uruchomiony webpack, szuka on katalogu "src" a w nim pliku "index.js")

Ten moduł wejściowy importuje inne moduły.

# entry (point/points)

Właściwość entry w eksportowanym obiekcie określa ścieżkę do pliku wejściowego. Podajemy ścieżkę względną względem katalogu głównego.

```
//webpack.config.js (czy inny plik konfiguracyjny)
```

```
module.exports = {  
  entry: './path/index.js' //względem katalogu głównego  
};
```

# Output

Na podstawie właściwości output bundler wie gdzie umieścić docelowe pliki. Domyślnie jest to ./dist/main.js dla pliku JavaScript i folder "dist" dla innych zasobów.

```
//webpack.config.js
module.exports = {
  entry: './path/index.js',
  output: {
    path: __dirname + '/dist', //wymaga ścieżki bezwzględnej
    filename: 'bundle.js'
  }
};
```

# Użycie innych modułów w webpack.config.js

Bardzo często w pliku konfiguracyjnym będziemy używali innych paczek.

Pamiętaj że sam plik konfiguracyjny to moduł Node.js (to nie jest plik bundlowany), więc wymaga obecnie użycia CommonJS i metody require().

```
//webpack.config.js
const path = require('path'); //Moduł Path, będący podstawowym modułem Node.js

module.exports = {
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js'
  }
};
```

# Nazwany plik(i) wejściowy i użycie nazwy w output

```
//webpack.config.js

module.exports = {
  //entry: './src/app.js',
  entry: {
    main: './src/app.js',
  },
  output: {
    filename: '[name].bundle.js',
    path: __dirname + '/dist')
  },
};
```



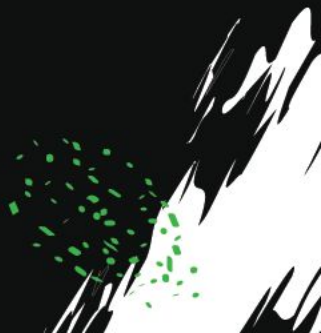
# Wartości domyślne

Pamiętajmy, że jeśli czegoś nie określimy w pliku konfiguracyjnym to przyjmuje to wartość domyślną i domyślnie tych wartości jest oczywiście mnóstwo. Jeśli nie określimy `output.path` to ta wartość wskazuje na `dist`.

```
module.exports = {  
  entry: './path/index.js',  
  output: {  
    //path: __dirname + '/build',  
    filename: 'bundle.js'  
  }  
};
```



# LOADERS



# Loader

Nie tylko moduły JavaScript (czy JSON) mogą być bundlowane przez webpack (one są obsługiwane domyślnie). Do innych typów plików potrzeba dodatkowych narzędzi w postaci loaderów (ładowarek, wczytywarek).

Loadery możesz traktować jako mini programy przekształcające inne typy plików w moduły.

By skorzystać z loadera, wymagana jest jego **instalacja** (`npm install nazwa-loadera --save-dev`) oraz dodanie go do konfiguracji (`webpack.config.js`) do właściwości **module.rules**. Loader do użycia nie wymaga zaimportowania do pliku konfiguracyjnego (nie używamy więc w tym przypadku `import` czy `require`).

Przykłady:

```
npm i file-loader --save-dev
```

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

# Loader - jak działają - trochę głębiej

Loadery pełnią rolę preprocesora. Działają na poziomie modułu (wskazujemy moduły obsługiwane przez dany/dane loadery). Dzięki loaderom możliwe jest praca nie tylko z plikami JavaScript i JSON. loadery dają **wiele możliwości przetworzenia modułów**.

Loader jest zaimplementowany jako funkcja, która jako argument przyjmuje przekazany kod źródłowy i która zwraca kod źródłowy po przetworzeniu.

Kod wejściowy np.

```
import './css/style.css'
```

(bez loaderów zgłosi błąd)

Loader np. css-loader

wczytuje przetwarza i zwraca

Możemy to sobie wyobrazić jako  
wywołanie funkcji css-loader(css)

Zapewnia możliwość odczytu

Loader np. style-loader

wczytuje przetwarza i zwraca

Możemy to sobie wyobrazić jako  
wywołanie funkcji style-loader(style)

Umieszcza w <Head> strony

# Loader - implementacja w projekcie - instalacja

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.txt$/,  
        use: 'raw-loader'  
      },  
      { test: /\.css$/, use: 'css-loader' },  
    ]  
  }  
};
```

# Loader - implementacja w projekcie - instalacja

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.txt$/,  
        use: 'raw-loader'  
      },  
      { test: /\.css$/, use: 'css-loader' },  
    ]  
  }  
};
```

# Loader - implementacja w projekcie - webpack.config.js

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.txt$/,  
        use: 'raw-loader'  
      }  
    ]  
  }  
};
```

Loadery dodajemy wewnątrz module.exports do **tablicy module.rules**. Tworzymy tutaj reguły dla konkretnych typów plików. We właściwości **test** określamy za pomocą **wyrażenia regularnego\*** o jaki plik nam chodzi (jakie ma rozszerzenie), a we właściwości **use** określamy jakie loadery mają być użyte (mogą być użyte więcej niż jeden).

\*Wyrażenie regularne jest wzorcem, którego poszukujemy w ciągu znaków. Wyrażenie regularne zawsze analizuje łańcuch znaków (stringa)

\$ - na końcu nazwy (pliku)

# wiele typów plików w jednej regule

```
module: {  
  rules: [  
    {  
      test: /\. (png|jpg|svg) $/,  
      use: [  
        'file-loader'  
      ]  
    }  
  ]  
},
```

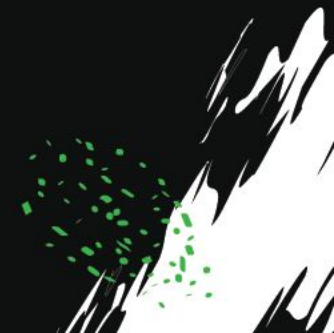




# od prawej do lewej - wiele loaderów dla danego pliku

```
module: {  
  rules: [  
    { test: /\.txt$/, use: 'raw-loader' },  
    { test: /\.css$/, use: ['style-loader', 'css-loader'] },  
    { test: /\.scss$/, use: ['style-loader', 'css-loader', 'sass-loader'] },  
    { test: /\.(png|jpg|svg)$/, use: ['file-loader'] },  
  ]  
},
```

Mamy tutaj do czynienia z łańcuchem funkcji na zasobie. Najpierw wykonana funkcja, która jest na końcu tablicy (ostatnia będzie pierwsza funkcja).



# Loader - alternatywny zapis use

//wersja 1 - nazwy loaderów w tablicy

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

//wersja 2 - w tablicy obiekty reprezentujące loadery

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

# Loader - jak działają - trochę głębiej

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

`style-loader(css-loader(sass-loader('plik.scss')))` - w taki sposób to działa

Pamiętaj, że najpierw wykonuje się funkcja najbardziej po prawo, ostatnia funkcja tablicy, a następnie kolejna funkcja w porządku od końca do początku tablicy.

# Plugins (wtyczki)

Pluginy zrobią to, czego nie są w stanie zrobić loadery.

O wtyczkach mówi się, że są kręgosłupem webpacka (bo webpack sam w sobie jest implementowany jako zbioru różnych pluginów).

Pluginy muszą zostać zainstalowane i w przeciwieństwie do loaderów, także zaimportowane w pliku konfiguracyjnym (webpack.config.js). W pliku konfiguracyjnym należy też stworzyć instancję danego pluginu.

# Plugins vs loadery

Loadery są preprocesem, można powiedzieć że wykonują się na początku kompilacji (bundlingu). Natomiast pluginy działają podczas całego procesu - mogą na początku (wspólnie z loaderami) czy pod koniec całego procesu bundlingu.

Loadery dotyczą pojedynczego moduły (pliku) - są wykonywane na module. Pluginy działają na całym generowanym pakiecie.

Loadery są funkcjami, pluginy udostępniają klasy/konstruktory czy interfejsy, których częścią jest klasa/konstruktor.

# Plugins (wtyczki) jako klasy/konstruktory

## HTML Webpack Plugin

Pluginy udostępnia klasę:

```
class HtmlWebpackPlugin { }
```

Ponieważ pluginy w webpacku udostępniają klasę/konstruktor, by ich użyć należy stworzyć ich instancję. Pluginy mogą także przyjmować różne argumenty - podajemy je przy tworzeniu ich instancji.

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

# Plugins (wtyczki)

```
//instalacja w projekcie jako devDependency
```

```
npm i --save-dev html-webpack-plugin
```

```
//plik konfiguracyjny webpacka - import
```

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

```
//plik konfiguracyjny webpacka - użycie jako element tablicy we właściwości plugins
```

```
module.exports = {
```

```
  plugins: [
```

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

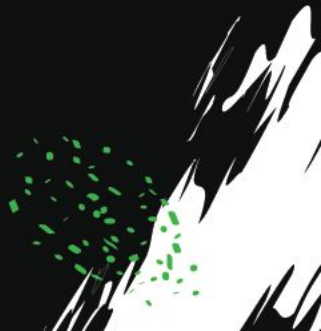
```
  ]
```

```
}
```



Wtyczka do czyszczenia katalogu dla bundlingu

# Clean Webpack Plugin





# Wtyczka do czyszczenia katalogu dla bundlingu

- instalacja

npm i -D clean-webpack-plugin

- użycie w pliku konfiguracyjnym

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

Więcej o pluginie:  
<https://github.com/johnagan/clean-webpack-plugin>

# webpack --watch

Do tej pory za każdym razem by zrobić bundlig musieliśmy wywołać skrypt w wierszu poleceń. Ale nie musimy tak robić, możemy skorzystać z trybu watch w webpacku.

Docelowo skorzystamy też z serwera deweloperskiego w webpacku, który oprócz opcji watch ma także m.in. automatyczne odświeżanie przeglądarki (póki co live servera w VSC).

```
"dev": "webpack --config ./config/webpack.config.js",  
"dev-watch": "npm run dev -- --watch",  
"watch": "webpack --watch",
```

# HASHING - Po co hashing?

Przeglądarka wie, że dopóki mamy tą samą nazwę pliku (dla wielu zasobów np. grafika, javascript), to dostarczamy jej taką samą zawartość. Przeglądarką uznaje więc, że nie musi danego zasobu ponownie pobierać, co oczywiście ma sens (są to zagadnienia związane z cachingiem / buforowaniem strony przez przeglądarkę).

Póki co dostarczaliśmy przeglądarce pliki o tej samej nazwie. Jeśli dostarczymy przy każdej zmianie zawartości bundlingu nową nazwę to **zmiana nazwy pliku wymusi na przeglądarce pobranie go ponownie**.

[hash]

```
PS D:\projekty\kursy\css-js\webpack\050922019> npx webpack --mode development
Hash: fe4518496d411a86af21
Version: webpack 4.39.3
Time: 738ms
Built at: 2019-09-06 12:46:57
```

Można dodać do nazwy bundlowanych paczek (plików) hash (identyfikator). Hash w nazwie będzie taki sam, dopóki nie zmieni się zawartość bundlingu. Jednak jeśli mamy wiele plików po bundlingu, to zmienią się wszystkie ich nazwy, chociaż nie wszystkie pliki musiały się zmieniać.

```
module.exports = {
  mode: 'development',
  entry: {
    main: './src/index.js',
  },
  output: {
    filename: '[hash].[name]-bundle.js',
    path: path.resolve(__dirname, '../', 'build')
  },
}
```

# [contenthash]

contenthash jest rekomendowanym rozwiązaniem dla naszych zbundlowanych plików. Możemy go też wykorzystywać w innych typach plików, choćby w css - jeśli zmieni się css, to będzie się zmieniała nazwa tylko zbundlowanego pliku CSS, a nie JavaScript.

Podobnie działa [chunkhash], ale rekomendowanym jest [contenthash]

```
module.exports = {  
  mode: 'production',  
  entry: {  
    main: './src/index.js',  
    app: './src/app.js',  
  },  
  output: {  
    filename: '[contenthash][name].js',  
    path: path.resolve(__dirname, '../', 'build')  
  },  
}
```

# [contenthash:4]

Możemy też wskazać wielkość hasha.

```
▼ build
JS 4a017cedba6914b15f97.main-bundle.js
JS 45727d1856e1d1a94a92.app-bundle.js
```

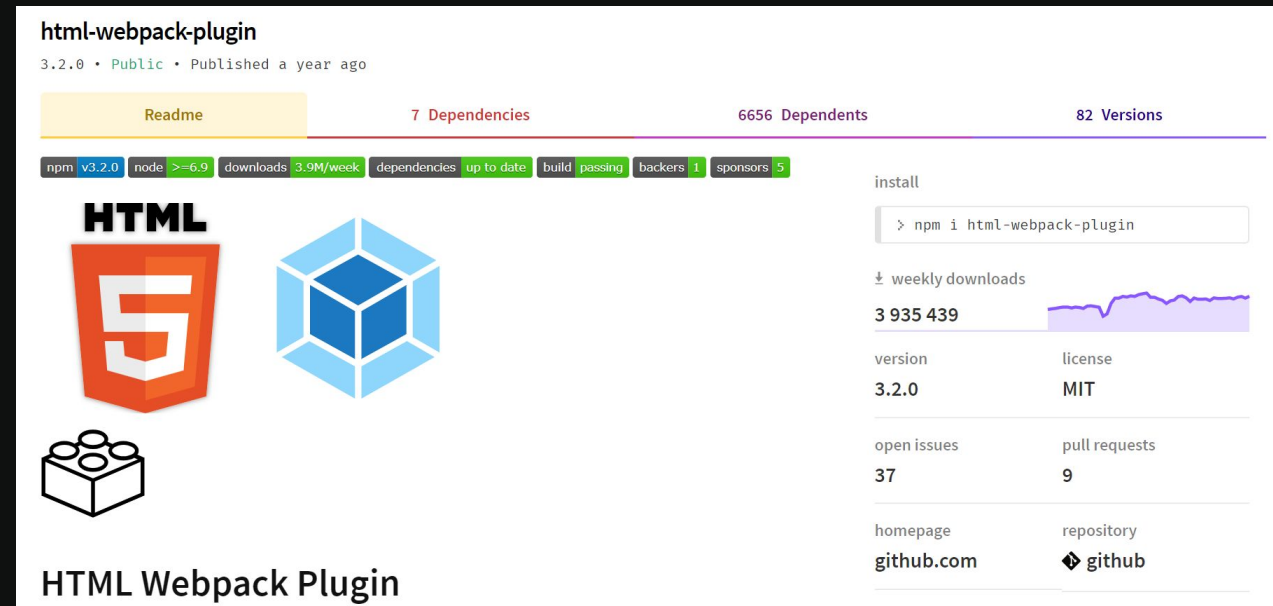
```
module.exports = {
  mode: 'development',
  entry: {
    main: './src/index.js',
    app: './src/app.js',
  },
  output: {
    filename: '[contenthash:6].[name]-bundle.js',
    path: path.resolve(__dirname, '../', 'build')
  },
}
```

```
▼ build
JS 5da812.app-bundle.js
JS eca6d9.main-bundle.js
```

# HTML - obsługa w webpacku

Zmienianie ścieżki (nazwy) pliku jest oczywiście bardzo złym rozwiązaniem. Zobaczmy jak uprościć pracę z plikiem czy plikami HTML w naszym projekcie.

Użyjemy do tego pluginu **HTML Webpack Plugin**



The screenshot shows the npm package page for **html-webpack-plugin**. At the top, it displays the package name, version 3.2.0, and its status as 'Public'. Below this, it shows '7 Dependencies', '6656 Dependents', and '82 Versions'. A row of badges indicates compatibility with npm v3.2.0, node >=6.9, 3.9M weekly downloads, up-to-date dependencies, passing builds, 1 backer, and 5 sponsors. The main content area features the 'HTML' logo, the Webpack logo, and a small icon of a brick. Below these is the text 'HTML Webpack Plugin'. On the right side, there is an 'install' section with a command box containing `> npm i html-webpack-plugin`. Below this is a 'weekly downloads' chart showing a peak of 3,935,439. A table lists package details: version 3.2.0, license MIT, 37 open issues, 9 pull requests, homepage github.com, and repository github.

property	value
version	3.2.0
license	MIT
open issues	37
pull requests	9
homepage	github.com
repository	github



# HTML - obsługa w webpacku




HTML Webpack Plugin pozwala nam też, dzięki licznym ustawieniom oraz dzięki możliwości pracy z różnym template'mi (szablonami jak pug czy handlebars), tworzyć strony składające się z wielu zróżnicowanych plików html.

**html-webpack-plugin**  
3.2.0 • Public • Published a year ago

Readme 7 Dependencies 6656 Dependents 82 Versions

npm v3.2.0 node >=6.9 downloads 3.9M/week dependencies up to date build passing backers 1 sponsors 5

**HTML**



HTML Webpack Plugin

install

```
> npm i html-webpack-plugin
```

↓ weekly downloads  
**3 935 439**

version	license
3.2.0	MIT
open issues	pull requests
37	9
homepage	repository
github.com	github



# Webpack Dev Server - serwer deweloperski

Webpack Dev Server - najpopularniejszy serwer deweloperski (nie służy do hostowania aplikacji/stron) do użycia z webpackiem.

Posiada tryb obserwowania, ale oprócz tego także automatycznie odświeża zawartość projektu w przeglądarce. Obsługuje także HMR (Hot Module Replacement), co pozwala na zmianę stanu bez odświeżania (jest to opcjonalne i przydatne np. w React, gdzie ponowne wczytywanie strony nie ma często sensu).

WDS przechowuje pliki w pamięci tzn. nie dokonuje bundlingu pakietów do wskazanego folder (nie zapisuje outputów do naszego katalogu).

WDS jest świetnym rozwiązaniem do developmentu i daje sporo możliwości konfiguracyjnych.

# Instalacja WDS

npm i webpack-dev-server -D

w skrypcie możemy napisać

```
"start": "webpack-dev-server --config path/webpack.config.js"
```

i uruchomić w wierszu poleceń: **npm start**

\* Domyślnie WDS korzysta z tego samego pliku konfiguracyjnego co webpack.

\*\* Możemy też po instalacji napisać bezpośredni w wierszu poleceń `npx webpack-dev-server` (i ewentualne opcje)

# Webpack dev server - konfiguracja

Za chwilę przyjrzymy się kilku parametrom konfiguracji naszego web serwera deweloperskiego.

Parametry określamy jako właściwości obiektu devServer w naszym pliku konfiguracyjnym.

```
module.exports = {  
  mode: 'development',  
  entry: {  
    main: './src/index.js',  
  },  
  output: {  
    filename: '[name]-bundle.js',  
  },  
  devServer: {  
  }  
}
```

# open - automatyczne otwieranie przeglądarki przy starcie webpack dev servera

```
devServer: {  
  open: true  
},
```

Możemy również napisać skrypt

```
"start": "webpack-dev-server --open"
```

lub bezpośrednio w wierszu poleceń

```
npx webpack-dev-server --open
```

Niezależnie od tego czy użyjemy --open czy nie, nasza aplikacja będzie dostępna domyślnie pod adresem

<http://localhost:8080/>

# contentBase - zasoby statyczne, których nie bundlujemy

```
devServer: {  
  // contentBase: false,  
  contentBase: path.resolve(__dirname, 'public')  
},
```

Content not from webpack is served from path\public (ta ścieżka powinna być bezwzględna stąd użyliśmy przy jej tworzeniu \_\_dirname).

Ścieżka w oparciu o to gdzie znajduje się plik konfiguracyjny.

*\* domyślnie szuka w folderze głównym. Jeśli damy false to nie będzie brał po uwagę takich zasobów*

Założmy bezpośredni dostęp do grafiki.

/public

image.jpg

/src

index.js

/build

index.html

34i32049i32.js

# port i host - możliwość ustawienia własnych

```
devServer: {  
  host: 'localhost', //domyślne  
  port: 5000, //domyślne 8080  
},
```



localhost:5000

# Wyświetlanie błędów

Błędy programu (niezwiązane z procesem bundlingu), możemy obejrzeć oczywiście w konsoli przeglądarki. Co ważne domyślnie (gdy mode jest ustawiony na development), zobaczymy też odwołanie do nazwy pliku przed bundlingiem.

```
✖ Uncaught ReferenceError: b is not defined
  at Module../src/message.js (message.js:8)
  at __webpack_require__ (bootstrap:19)
  at Module../src/index.js (index.js:1)
  at __webpack_require__ (bootstrap:19)
  at Object.0 (main-3d47fbd....js:9942)
  at __webpack_require__ (bootstrap:19)
  at bootstrap:83
  at bootstrap:83
```

message.js:8

[WDS] Live Reloading enabled.

client:52



Wprowadzenie i podstawowa konfiguracja za nami!

