Webpack

What is webpack

Web application contains many resources:

- images - jpg, png, bmp
- styling - css, sass, scss, less
- font files
- script files - Vanilla JS, ES6, TypeScript, Coffee Script…
- HTML

…
Those different resources need different handling, for example image files we will need to optimize and reduce their size,
some styling files we will have to compile to css files like SASS, SCSS, LESS.
Some script files we might want to interpret to Vanilla JS like ES6 or TypeScript.
HTML files we might want to concat to reduce size.
Webpack helps us mash up all our application resources, perform the needed action for each resource and output a packed web application with minimal resources.
We can also use webpack to push those static resource to our CDN.
Webpack is an essential tool for processing our web application files and ease our way into deploying our application.

How webpack works

We are giving webpack a configuration file which has an **entry** property which tells webpack what are the files to start from. from the starting files webpack is recursively building dependency graphs which includes all the modules our application needs.
then webpack will process each module according to what is specified in the configuration **Loaders** section, and will bundle all the modules to the minimal files needed.

Installing webpack

Installing webpack is done via **npm.**
We will install it locally with every local project we want to use webpack with.
Let's start a new project and open a directory called **webpack-sample**

> mkdir webpack-sample

> cd webpack-sample

We will start a local npm package installation in this folder. In your **CMD** or **Terminal** type:

> npm init --yes

A **package.json** file will be created with the default options.
Now to install **webpack** in a **node_modules** directory inside the current **webpack-sample** directory, we just type in the **Terminal:**

> npm install webpack --save-dev

notice that we are putting the **--save-dev** flag in order to save the package we installed in the **package.json** file, also notice that **webpack** is a dev dependency and will not be required if a user is using the package you are just creating.

What we will try to do with webpack

In this lesson we will make **webpack** process a **TypeScript** file, transform the file to a **JS** file and minify and uglify the file to reduce it's size.

Configuration file

Webpack required a configuration file to know how to work, and unless specified it will look for a file called: **webpack.config.js.** You can also change the name of the config file by passing the **--config** flag when running **webpack** commands.
lets create a configuration file, open a text editor of your choice and in the **webpack-sample** directory, create a file called: **webpack.config.js.**
a webpack configuration file is a node packages with a key value dictionary, so a configuration file starts with:

```
module.exports = {
    // webpack configuration key value dictionary
    // ...
}
```

Let's start going over the basic settings of a webpack configuration

Entry

As we mentioned, webpack creates a graph of dependency, the root of the graph is set by the entry configuration. We can also place multiple entries to create multiple graph that are independent from each other.
The entry key will accept a string an array or a dictionary and in most cases we will place a string for a single entry point and an object for multiple entry points.
To create a single entry point to a file called **main.ts** that is located in the same directory as the config file, we can place the following code:

```
module.exports = {
    entry: './main.ts'
}
```

to make **webpack** process 2 files a **JS** file and a **TypeScript** file, we can place a dictionary as the **entry** value like this:

```
module.exports = {
    entry: {
        main: './main.ts',
        vendor: './vendor.js'
    }
}
```

since our goal is to process a **TypeScript** file let's keep the entry as the first value so your configuration should now look like so:

```
module.exports = {
    entry: './main.ts'
}
```

Webpack will search for the file **main.ts** so with your **IDE** care a file called **main.ts** with the following code:

```
class Pokemon {
    constructor(public name : string = ''){}

    sayMyName() {
        console.log(`Hello my name is: ${this.name}`);
    }
}
```

So with the entry we can set the root of the graph that webpack is building for our app.

Output

Webpack will create a dependency graph according to the **entry** in our configuration file, it will then go over the graph from the root to the entire resources of our application and it will process the files. For example it will take our **TS TypesScript** files and transform them to **Vanilla JS** it will take the **SCSS** and compile them to **CSS.** The output option in the configuration will instruct **webpack** where to place the processed files, which directory, which filename and what URL address they will have.
This option excepts and object and the important keys in that object are:

- **filename -** what name should i call the files that I process, if we have multiple files we can use placeholders that **webpack** will replace like **[name]** or **[hash]**
- **path** - the absolute path where webpack will place the files that it process
- **publicPath -** in some cases or plugins, webpack will require to place the url to the resource. If we are placing the files in a **CDN** we can use this option to tell webpack what url the files will have.

In our example we are outputting a single file that turns our **typescript** file to **VanillaJs** and we don't have a **CDN** so our output configuration will be simple. Change your **webpack.config.js** to look like this:

```
var path = require('path');
module.exports = {
	entry: './main.ts',
	output: {
		filename: 'main.js',
		path: path.resolve(__dirname, 'dist')
	}
}
```

notice that the **path** get's and absolute path so we can use **path** and **__dirname** to get the absolute path of the configuration file and create a **dist** folder.

Loaders

Loaders instruct **webpack** in which way to deal with each file we have in our dependency graph. Loaders are installed via **npm**.
For example in this example we want to transform our **typescript** file to a **js** file, so we will use the **ts-loader**.
each loader is an object with the following key's:

- **test -** the regular expression that associate a file of type to this loader
- **loader/use -** string of the loader to use

- **options -** additional options this loader has

Loaders can also be chained by using the **!** mark for example if we have a **scss** files we might want to chain the loaders that deal with them to first be the **scss-loader** and then be the **css-loader** the **loader** config for this might look like this:

```
…
module: {
        loaders: [
                {test: /\.scss$/, use: ‘css-loader!scss-loader’}
        ]
}
...
```

notice that the direction that the file is processed if from the right to left, first the **scss-loader** and then the **css-loader**
In our example we want to associate files with **ts** extension to the **typescript** loader.
first let’s install **typescript** by typing in the terminal:

> npm install typescript --save-dev

Now let’s install the **ts-loader** by typing:

> npm install ts-loader --save-dev

TypeScript will also need a configuration file called **tsconfig.js.** Let’s create this file by typing in the terminal:

> tsc --init

Now modify the configuration file to look like this:

```
var path = require('path');
module.exports = {
      entry: './main.ts',
      output: {
            filename: 'main.js',
            path: path.resolve(__dirname, 'dist')
      },
      module: {
            loaders: [
                  {test: /\.ts$/, use: 'ts-loader'}
```

```
        ]
    }
}
```

We just assigned the **ts-loader** to our typescript files.

Plugins

While loader process a file, a plugin can perform an action on entire compilations or chunks.
In order to use a plugin you need to pass an instance of the plugin in the **plugins** array of the configuration file.
Here is a list of some of the plugins that are commonly used in a lot of applications:

- **html-webpack-plugin -** we can use webpack to help us with caching of files by adding **[hash]** placeholder in the **output.filename** configuration. Since we need to create a proper **HTML** file which will load our file and the hash is determined on compilation time, we can use this plugin to help us generate the proper HTML which loads the file that we are creating.
- **UglifyJsWebpackPlugin -** will concat and minify our code.
- **ExtractTextWebpackPlugin -** will output all the **require** or **import** of css files to a single external file
- **webpack-s3-plugin -** will upload the compiled files to **S3** where you can set your **CDN** to serve files from there.

In our simple example let's set the **Uglify** plugin to minimize our file.
Modify you **webpack.config.js** to look like this:

```
var path = require('path');
var webpack = require('webpack');
module.exports = {
    entry: './main.ts',
    output: {
        filename: 'main.js',
        path: path.resolve(__dirname, 'dist')
    },
    module: {
        loaders: [
            {test: /\.ts$/, use: 'ts-loader'}
        ]
    },
    plugins: [
        new webpack.optimize.UglifyJsPlugin()
    ]
}
```

Compiling with Webpack

Now based on the configuration we just created, let's run webpack to turn our **TypeScript** file to **JS** file.
In your terminal type:

> webpack

You should see that the file **main.js** is created for us in the **dist** folder and also that this file is minified.
the command above will run **webpack** a single time, if you want **webpack** to rerun on every change in the dependency graph you can run on watch mode with the command:

> webpack -w

Student EX.

Try and use **webpack** to transfrom a **scss** file to a **css** file.

Conclusion

webpack is a must tool for web applications with a lot of resources to deal with, it's really easy to understand and far superior to other alternative solutions like **Grunt** or **Gulp**.