FREE eBook

LEARNING webpack

Free unaffiliated eBook created from
Stack Overflow contributors.

#webpack

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: webpack

It is an unofficial and free webpack ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official webpack.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with webpack

Remarks

Webpack is a module bundler which reads modules with dependencies and produces static assets representing those modules.

It features an extendable *loader system* which allows bundles to include not only Javascript assets, but CSS, Images, HTML and much more.

For example, using the in-built Javascript loader, css-loader and url-loader:

```
require("./code.js") // Load Javascript dependency
var css = require("./styles.css"); // Load CSS as a string
var base64Image = require("./image.png"); // Load an image as a base64 string
```

Would become a single bundled file:

```
// From code.js
console.log("Hello, World");
// From styles.css
var css = "body { margin: 0; padding: 0; } h1 { color: #FF0000; }";
// From image.png
var base64Image =
"
```

Dependencies can be defined in any of the most common module styles (CommonJS & AMD).

Versions

Version	Release date
3.0.0	2017-06-19
2.6.1	2017-05-25
2.6.0	2017-05-23
2.5.1	2017-05-07
2.5.0	2017-05-04
2.4.1	2017-04-14
2.4.0	2017-04-14
1.13	2016-04-17

Version	Release date
1.12	2015-08-25
1.11	2015-08-06
1.10	2015-06-27
1.9	2015-05-10
1.8	2015-04-29
1.7	2015-03-11
1.6	2015-02-24
1.5	2015-01-21
1.4	2014-12-28
1.3	2014-08-25
1.2	2014-05-27
1.1	2014-05-17
1.0	2014-03-01
0.11	2013-12-31
0.10	2013-06-19
0.9	2013-03-19
0.8	2013-01-21

Examples

Installation

Prerequisites:

NodeJS and npm

There are two ways of installing Webpack: globally or per-project. It is best to have the dependency installed per-project, as this will allow you to use different versions of webpack for each project and don't require user to have installed webpack globally.

Installing per-project

Run the following command from the root folder of your project:

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

You can then run the webpack executable installed to `node_modules`:

```
./node_modules/.bin/webpack
```

Or create an NPM script in your `package.json` file, where you can omit the `node_modules` part - npm is smart enought to include that folder in its PATH.

```
// in package.json:
{
  ...
  "scripts": {
    "start": "webpack"
  },
  ...
}

// from terminal:
npm start
```

Installing globally

Run the following command at a prompt:

```
npm install webpack -g
```

Example of webpack.config.js with babel

Dependencies

```
npm i -D webpack babel-loader
```

webpack.config.js

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

module.exports = {
  entry: {
    app: ['babel-polyfill', './src/'],
  },
  output: {
    path: __dirname,
    filename: './dist/[name].js',
  },
  resolve: {
    extensions: ['', '.js'],
  },
  module: {
    loaders: [{
```

```
      test: /\.js$/,
      loaders: ['babel-loader'],
      include: path.resolve(__dirname, 'src')
    }],
  }
};
```

Example of Javascript + CSS + Fonts + Images

Required modules

```
npm install --save-dev webpack extract-text-webpack-plugin file-loader css-loader style-loader
```

Folder structure

```
.
└── assets
    ├── css
    ├── images
    └── js
```

webpack.config.js

```
const webpack = require('webpack');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const path = require('path');
const glob = require('glob');

module.exports = {
  entry: {
    script: path.resolve(__dirname, './assets/js/app.js'),
    style: path.resolve(__dirname, './assets/css/app.css'),
    images: glob.sync(path.resolve(__dirname, './assets/images/**/*.*')),
  },
  context: __dirname,
  output: {
    path: path.resolve('./dist/assets'),
    publicPath: '/dist/assets',
    filename: '[name].js',
  },
  module: {
    loaders: [
      {
        test: /\.css$/,
        loader: ExtractTextPlugin.extract({
            fallback: 'style-loader',
            use: 'css-loader'
        }),
      },
      {
        test: /(\.woff2?|\.woff|\.ttf|\.eot|\.svg)(\?v=\d+\.\d+\.\d+)?$/,
        loader: 'file-loader?name=[name]-[hash:6].[ext]',
      },
      {
        test: /\.(png|jpe?g|gif|ico)$/,
        loader: 'file-loader?name=[name].[ext]',
```

```
      },
    ],
  },
  plugins: [
    new ExtractTextPlugin('app.css' /* optional: , { allChunks: true } */),
  ],
};
```

`glob.sync('./assets/images/**/*.*')` will require all files in the images folder as entry.

`ExtractTextPlugin` will grab the generated output and create a bundled `css` file.

Webpack Simple Example

The minimum required to use Webpack is the following command:

```
webpack ./src/index.js ./dist/bundle.js

// this is equivalent to:

webpack source-file destination-file
```

Web pack will take the source file, compile to the output destination and resolve any dependencies in the source files.

Webpack, React JSX, Babel, ES6, simple config

Ensure that you install the correct npm dependencies (babel decided to split itself into a bunch of packages, something to do with "peer dependencies"):

```
npm install webpack webpack-node-externals babel-core babel-loader babel-preset-react babel-preset-latest --save
```

`webpack.config.js`:

```
module.exports = {
    context: __dirname, // sets the relative dot (optional)
    entry: "./index.jsx",
    output: {
        filename: "./index-transpiled.js"
    },
    module: {
        loaders: [{
            test: /\.jsx$/,
            loader: "babel?presets[]=react,presets[]=latest" // avoid .babelrc
        }]
    }, // may need libraryTarget: umd if exporting as a module
    externals: [require("webpack-node-externals")()], // probably not required
    devtool: "inline-source-map"
};
```

`webpack-node-externals` is a function that scans your `node_modules` and ensures that they aren't transpiled and bundled along with your front-end code, though it ensures the bundle retains reference to them. This helps with faster transpilation, since you're not re-encoding libraries.

Simple webpack setup with Node.js

Folder Structure

```
.
├── lib
├── modules
|   ├── misc.js
|   ├── someFunctions.js
├── app.js
├── index.html
├── package.json
├── webpack.config.js
└── webserver.js
```

package.json

```
{
  "name": "webpack-example-with-nodejs",
  "version": "1.0.0",
  "description": "Example using webpack code-splitting with some Node.js to support the
example",
  "main": "webserver.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "@Gun",
  "license": "ISC",
  "devDependencies": {
    "body-parser": "^1.17.1",
    "express": "^4.15.2",
    "http": "0.0.0",
    "morgan": "^1.8.1",
    "multer": "^1.3.0",
    "webpack": "^2.4.1"
  }
}
```

webpack.config.js

```
var path = require('path'); // used to get context

module.exports = {
    context: path.join(__dirname, 'app'), // resolves entry below, must be absolute path
    entry: './app.js', // entry point or loader for the application
    output: {
        path: path.join(__dirname, 'app/lib'), // express static folder is at /app/lib
        filename: '[name].bundle.js', // the file name of the bundle to create.  [name] is
replaced by the name of the chunk (code-splitting)
        publicPath: 'static' // example uses express as the webserver
    }
};
```

webserver.js

```
var express = require('express'),
```

```
    path = require('path'),
    bodyParser = require('body-parser'),
    multer = require('multer')()
    logger = require('morgan'),
    fs = require('fs'),
    http = require('http');

var app = express();
var port = 31416;

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(logger('short'));
app.use('/jsBundles',express.static('lib'));
app.get('/', function(request, response){
    response.sendFile(__dirname + '/index.html');
});

var server = http.createServer(app).listen(port, function(){
    console.log("I feel a disturbance in the port:" + port);
});
```

index.html

```
<!DOCTYPE html>
<html>
    <body>
        <div id="someValue"><label for="num">Enter a number:</label><input id="num" /></div>
        <div class="buttonList">
            <ul>
                <li><button id="doubleIt">double it</button></li>
                <li><button id="tripleIt">triple it</button></li>
            </ul>
        </div>
        <div id="someOtherValue">
            And the count shall be: <span id="theCount"></span>
        </div>
        <script src="/jsBundles/main.bundle.js"></script>
    </body>
</html>
```

app.js

```
require(['./modules/someFunctions'],function(){
        window.onload = function(){
                var someFunctions  = require('./modules/someFunctions');
                document.getElementById('doubleIt').onclick = function(){
                        var num = document.getElementById('num').value;
                        document.getElementById('theCount').innerHTML =
someFunctions.Double(num);
                };

                document.getElementById('tripleIt').onclick = function(){
                        var num = document.getElementById('num').value;
                        document.getElementById('theCount').innerHTML =
someFunctions.Triple(num);
                };
        };
});
```

misc.js

```
var self = {};
self.isNumber = function(value){
    // http://stackoverflow.com/questions/9716468/is-there-any-function-like-isnumeric-in-
javascript-to-validate-numbers
    return !isNaN(parseFloat(value)) && isFinite(value);
};
module.exports = self;
```

someFunctions.js

```
require(['./misc'], function(){
    var misc= require('./misc');

    var self = {};
    self.Double = function(value){
        if(!misc.isNumber(value)){
            return 0;
        };
        return value*2;
    }

    self.Triple = function(value){
        if(!misc.isNumber(value)){
            return 0;
        };
        return value*3;
    }

    module.exports = self;
});
```

NOTE

run ***npm i --save-dev*** to install dependencies

run ***node .\node_modules\webpack\bin\webpack.js*** once dependencies are installed

run ***node webserver.js*** to start the server

Read Getting started with webpack online: https://riptutorial.com/webpack/topic/924/getting-started-with-webpack

Chapter 2: Development server: webpack-dev-server

Examples

Installation

`webpack-dev-server` can be installed via `npm`

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

now you can start server

```
./node_modules/.bin/webpack-dev-server
```

To simplify usage you can add script to `package.json`

```
// package.json
{
  ...
  "scripts": {
    "start": "webpack-dev-server"
  },
  ...
}
```

now to run server you can use

```
npm run start
```

`webpack-dev-server` is configured in `webpack.config.js` file in section `devServer`.

To change server content base directory you can use option `contentBase`. Example configuration setting root directory to `public_html` could look like

```
let path = require("path");

module.exports = {
  ...
  devServer: {
    contentBase: path.resolve(__dirname, "public_html")
  },
  ...
}
```

Using proxy

`webpack-dev-server`

can proxy some requests to others servers. This might be useful for developing API client when you want to send requests to same domain.

Proxy is configured via `proxy` parameter.

Example configuration of dev server passing requests to `/api` to other service listening on port 8080 might look like this

```
// webpack.config.js
module.exports = {
  ...
  devServer: {
    proxy: {
      "/api": {
        target: "http://localhost:8080"
      }
    }
  }
...
}
```

rewrite

It is possible to rewrite destination path using `pathRewrite` option.

Assuming you want to strip `/api` prefix from previous example your config might look like

```
// webpack.config.js
  ...
  devServer: {
    proxy: {
      "/api": {
        target: "http://localhost:8080",
        pathRewrite: {"^/api" : ""}
      }
    }
  }
...
```

Request `/api/user/256` will be converted to `http://localhost:8080/user/256`.

filter

It is possible to proxy only some requests. `bypass` allows you to provide function which return value will determine if request should be proxied or not.

Assuming you only want to proxy only POST requests to `/api` and let `webpack` handle the rest your configuration might look like this

```
// webpack.config.js
```

```
  ...
  devServer: {
    proxy: {
      "/api": {
        target: "http://localhost:8080",
        bypass: function(req, res, proxyOptions) {
          if(req.method != 'POST') return false;
        }
      }
    }
  }
...
```

Read Development server: webpack-dev-server online:
https://riptutorial.com/webpack/topic/9877/development-server--webpack-dev-server

Chapter 3: DllPlugin and DllReferencePlugin

Introduction

The Dll and DllReference plugins allow the code to be split in multiple bundles in a way the bundles can be compiled independently.

It is possible to build "vendor" scripts in a library that does not need to be compiled often (ex: React, jQuery, Bootstrap, Fontawesome...) and reference it in your app bundle that will need those scripts.

The application bundle, the one that is constantly going to be changed, will be in a separate configuration just referencing a already built "vendor" bundle.

Syntax

- new webpack.DllPlugin({ path: '[name]-manifest.json', name: '[name]_[hash]' })
- new webpack.DllReferencePlugin({ context: __dirname, manifest: require('./packname-manifest.json') })

Examples

Vendor configuration (DllPlugin)

Note: The `output.library` and `name` (in DllPlugin) must be the same.

```
const path = require('path');
const webpack = require('webpack');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const extractCSS = new ExtractTextPlugin('vendor.css');
const isDevelopment = process.env.NODE_ENV !== 'production';

module.exports = {
  resolve: {
    extensions: ['.js'],
  },
  module: {
    rules: [
      { test: /\.(png|woff|woff2|eot|ttf|svg)$/, loader: 'url-loader?limit=100000' },
      { test: /\.s?css$/i, loader: extractCSS.extract(['css-loader?minimize', 'sass-loader'])
},
      { test: /\.json$/, loader: 'json-loader' },
    ],
  },
  entry: {
    vendor: [
      'babel-polyfill',
      'font-awesome/scss/font-awesome.scss',
      'bootstrap/scss/bootstrap.scss',
      'jquery',
```

```
      'history',
      'react',
      'react-dom',
      'redux',
      'react-redux',
      'react-router',
      'react-router-dom',
      'react-router-redux',
      'redux-thunk',
    ],
  },
  output: {
    path: path.resolve('./dist'),
    filename: '[name].js',
    library: '[name]_[hash]',
  },
  plugins: [
    extractCSS,
    new webpack.DllPlugin({
      path: path.join(__dirname, 'dist', '[name]-manifest.json'),
      name: '[name]_[hash]',
    })
  ].concat(isDevelopment ? [] : [
    new webpack.optimize.UglifyJsPlugin({
      beautify: false,
      comments: false,
    }),
  ]),
};
```

Referencing a Dll Bundle (DllReferencePlugin)

Note: `manifest` (in DllReferencePlugin) should reference `path` (defined in DllPlugin)

```
const webpack = require('webpack');
const path = require('path');
const isDevelopment = process.env.NODE_ENV !== 'production';

const ExtractTextPlugin = require('extract-text-webpack-plugin');
const extractCSS = new ExtractTextPlugin('app.css');

const merge = require('extendify')({ isDeep: true, arrays: 'concat' });

module.exports = merge({
  context: __dirname,
  entry: {
    app: (isDevelopment ? ['webpack-hot-middleware/client'] : []).concat(['./src/']),
  },
  output: {
    path: path.resolve('./dist'),
    publicPath: '/static',
    filename: '[name].js',
  },
  resolve: {
    extensions: ['.js', '.ts', '.tsx'],
  },
  module: {
    loaders: [
      {
```

```
        test: /\.tsx?$/,
        loader: 'babel-loader!awesome-typescript-loader?forkChecker=true',
        include: /src|spec/,
      },
      {
        test: /\.s?css$/,
        loader: extractCSS.extract(['css-loader?minimize', 'sass-loader']),
        include: /src/,
      },
    ],
  },
  plugins: [
    new webpack.DllReferencePlugin({
      context: __dirname,
      manifest: require('./dist/vendor-manifest.json'),
    }),
    new webpack.DefinePlugin({
      'process.env': {
        'ENV': JSON.stringify(process.env.NODE_ENV),
      },
    }),
    extractCSS,
  ],
}, isDevelopment ? require('./webpack.config.development') :
require('./webpack.config.production'));
```

Read DllPlugin and DllReferencePlugin online: https://riptutorial.com/webpack/topic/9508/dllplugin-and-dllreferenceplugin

Chapter 4: Hot Module Replacement

Remarks

webpack-hot-middleware

Use with `webpack-dev-middleware`, by adding `webpack-hot-middleware/client` to entry.

Config

Add configs as query string to the path. Example:

```
webpack-hot-middleware/client?path=/__what&timeout=2000&overlay=false
```

Option	Description
path	The path which the middleware is serving the event stream on
timeout	The time to wait after a disconnection before attempting to reconnect
overlay	Set to false to disable the DOM-based client-side overlay.
reload	Set to true to auto-reload the page when webpack gets stuck.
noInfo	Set to true to disable informational console logging.
quiet	Set to true to disable all console logging.
dynamicPublicPath	Set to true to use webpack publicPath as prefix of path. (We can set `__webpack_public_path__` dynamically at runtime in the entry point, see note of `output.publicPath`)

Examples

Use with webpack-dev-middleware

1. Install *webpack-dev-middleware* via npm

   ```
   npm i -D webpack-dev-middleware webpack-hot-middleware
   ```

2. Modify *webpack.config.js*

   - Add `webpack-hot-middleware/client` to each items defined in `"entry"`

- Add `new webpack.HotModuleReplacementPlugin()` to `"plugins"`

```
module.exports = {
    entry: {
        js: [
            './index.js',
            'webpack-hot-
middleware/client?path=/__webpack_hmr&timeout=20000&reload=true'
        ]
    },
    plugins: [
        new webpack.HotModuleReplacementPlugin()
    ]
};
```

3. Add these to *index.js*

```
var webpack = require('webpack');
var webpackDevMiddleware = require('webpack-dev-middleware');
var webpackHotMiddleware = require('webpack-hot-middleware');

var config = require('./webpack.config.js');
var compiler = webpack(config);

app.use(webpackDevMiddleware(compiler, {
    noInfo: true,
    publicPath: config.output.publicPath,
    stats: { colors: true },
    watchOptions: {
        aggregateTimeout: 300,
        poll: true
    },
}));

app.use(webpackHotMiddleware(compiler, {
    log: console.log,
}));
```

Enable HMR for Module

To make a module eligible for Hot Module Replacement (HMR), the simplest way is to add `module.hot.accept()` inside the module, like this:

```
// ...

if(module.hot) {
    module.hot.accept(); // This will make current module replaceable
}
```

Use with webpack-dev-server

1. Install *webpack-dev-server* via npm.

```
npm i -D webpack-dev-server
```

2. Configure `webpack-dev-server` by adding *server.js*.

```
// server.js

var webpack = require("webpack");
var WebpackDevServer = require("webpack-dev-server");
var config = require("./webpack.dev.config");

var server = new WebpackDevServer(webpack(config), {
    // ...
});

server.listen(8080);
```

3. Modify *webpack.config.js*

- Add `webpack-dev-server/client` to each items defined in `"entry"`.
- Add `webpack/hot/only-dev-server` to each items defined in `"entry"`.
    - **NOTE:** Change if needed...
    - Use `webpack/hot/only-dev-server` to block page refresh if HMR fails.
    - Use `webpack/hot/dev-server` to auto-refresh page if HMR fails.
- Add `new webpack.HotModuleReplacementPlugin()` to `"plugins"`

```
module.exports = {
    entry: {
        js: [
            'webpack-dev-server/client?http://localhost:8080'
            'webpack/hot/only-dev-server',
            './index.js'
        ]
    },
    plugins: [
        new webpack.HotModuleReplacementPlugin()
    ]
};
```

4. Add `hot: true` in `webpack-dev-server` configuration

```
var server = new WebpackDevServer(webpack(config), {
    hot: true

    // ... other configs
});
```

Read Hot Module Replacement online: https://riptutorial.com/webpack/topic/4594/hot-module-replacement

Chapter 5: Loaders

Remarks

Webpack loaders can be configured as "preLoaders", "loaders" and "postLoaders". Although they don't have to be, configurations which use linting or other imperative or serial operations can take advantage of these build stages in the pipeline.

The key to understanding loaders and their use is that Webpack will run each module in the require graph through the loader system. Following the example above, this means that as Webpack begins crawling through the imports of your application, it will identify the files required and using a simple regex, will determine which file or file type requires what loader or series of loaders.

Above you can see that all ".js" or ".jsx" files will be es-linted by the eslint-loader in the preLoader phase. Other `js|x` file types will also be run through the babel-loader in the main loader phase. Also, in the same phase, any `.scss` files will be loaded into the sass-loader. This allows you to import Sass files in your JS modules and have them be output to the resulting JS bundle or even another separate standalone CSS bundle (using a plugin).

Note: Importing `.scss` files will only work with Webpack and an appropriate loader. Node will not understand this kind of import without a pre-processor or transpiler.

Also of note in the `.scss` example is the ability to "chain" loaders using the `!` exclamation mark as a "pipe" between different loaders. The example above pipes the output of the "sass-loader" into the "css-loader" and finally to the "style-loader" This could also be performed with an array of `loaders: ['style-loader', 'css-loader', 'sass-loader']`. Different options are also available to inline loader definitions and follow the query parameter syntax commonly found in URLs.

See also: https://webpack.github.io/docs/loaders.html

Examples

Config using preLoader for eslint, babel for jsx and css loader chaining.

The following configuration can be used as a base config for bundling up your project as a library. Notice how the module config contains a list of preLoaders and loaders.

```
// webpack.config.js

var path = require('path');

module.exports = {
    entry: path.join(__dirname, '..', 'src/index.js'),
    output: {
        path: path.join(__dirname, '..', '/lib'),
        filename: outputFile,
```

```
        library: 'myCoolBundle.js',
        libraryTarget: 'umd',
        umdNamedDefine: true
    },
    module: {
        preLoaders: [
            {
                test: /(\.jsx|\.js)$/,
                loader: "eslint-loader",
                exclude: /node_modules/,
            }
        ],
        loaders: [
            {
                test: /(\.jsx|\.js)$/,
                loader: ['babel'],
                exclude: /(node_modules)/,
                include: path.join(__dirname, '..'),
                query: {
                    presets: [ 'es2015', 'react']
                }
            },
            {
                test: /\.scss$/,
                loaders: ["style-loader", "css-loader!sass-loader"]
            }
        ]
    },
    resolve: {
        root: path.resolve(__dirname, '..', './src'),
        extensions: ['', '.js', '.jsx', '.scss'],
        fallback: path.join(__dirname, '../node_modules')
    },
    eslint: {
        configFile: path.resolve(__dirname, '..', '.eslintrc'),
    }
};
```

Read Loaders online: https://riptutorial.com/webpack/topic/6534/loaders

Chapter 6: Loaders & Plugins

Remarks

Loaders and plugins make up the building blocks of Webpack.

Loaders are typically delegated to a single task and file type. They are easier to setup and usually require less boilerplate code.

Plugins, on the other hand, have access to Webpack's internal build system via hooks, and are therefore more powerful. Plugins can modify the fully configured Webpack environment, and they can perform custom actions throughout the compilation process.

When dealing with our CSS files, for example, a loader might be used to automatically add vendor prefixes to properties, while a plugin might be used to produce a minified stylesheet in the bundler build process.

Examples

Getting started with loaders

To begin, `npm install` the desired loaders for your project.

Inside of the configuration object that is being exported in `webpack.config.js`, a `module` property will hold all of your loaders.

```
const source = `${__dirname}/client/src/`;

module.exports = {
  // other settings here

  module: {
    loaders: [
          {
            test: /\.jsx?$/,
            include: source,
            loaders: ['babel?presets[]=es2015,presets[]=react', 'eslint']
          },
          {
            test: /\.s?css$/,
            include: source,
            loaders: ['style', 'css', 'autoprefixer', 'sass']
          }
        ]
  },
};
```

In the above configuration, we're using three basic settings for our loaders:

- **test:** This is where we bind loaders to specific extensions using RegExp. The first set of

loaders is being executed on all .js and .jsx files. The second set is being executed on all .css and .scss files.

- **include:** This is the directory we want our loaders to run on. Optionally, we could just as easily use the `exclude` property to define directories we do not want included.
- **loaders:** This is a list of all the loaders we want to run on the files specified in `test` and `include`.

It's important to note that loaders are executed from right to left in each loaders array, and from bottom to top in the individual definitions. The code below will execute the loaders in the following order: `sass, autoprefixer, css, style`.

```
loaders: [
  {
    test: /\.s?css$/,
    include: source,
    loaders: ['style', 'css', 'autoprefixer']
  },
  {
    test: /\.s?css$/,
    include: source,
    loaders: ['sass']
  }
]
```

This is a common source of confusion and bugs for developers who are new to webpack. For example, when using JSX and ES6 syntax, we want to lint that code, *not* lint the compiled code that is provided by our babel loader. Therefore, our eslint loader is placed to the right of our babel loader.

The `-loader` suffix is optional when listing our loaders.

```
loaders: ['sass']
```

... is equivalent to:

```
loaders: ['sass-loader']
```

Alternatively, you can use the `loader` property (singular) along with a string separating the list of loaders by the `!` character.

```
loaders: ['style', 'css']
```

... is equivalent to:

```
loader: "style!css"
```

loading typescript files

To use typescript with webpack you need `typescript` and `ts-loader` installed

```
npm --save-dev install typescript ts-loader
```

Now you can configure webpack to use typescript files

```
// webpack.config.js

module.exports = {
  ..
  resolve: {
    // .js is required for react imports.
    // .tsx is required for react tsx files.
    // .ts is optional, in case you will be importing any regular ts files.
    extensions: ['.js', '.ts', '.tsx']
  },
  module: {
    rules: [
      {
        // Set up ts-loader for .ts/.tsx files and exclude any imports from node_modules.
        test: /\.tsx?$/,
        loaders: isProduction ? ['ts-loader'] : ['react-hot-loader', 'ts-loader'],
        exclude: /node_modules/
      }
    ]
  },
  ...
};
```

Read Loaders & Plugins online: https://riptutorial.com/webpack/topic/5651/loaders---plugins

Chapter 7: Tree Shaking

Examples

ES2015 tree shaking

webpack 2 introduces tree shaking which can remove unused code when ES2015 modules are used to import and export code.

Install

```
npm install babel-preset-es2015-webpack --save-dev
```

Usage

in .babelrc:

```
{
    "presets": [
        "es2015-webpack"
    ]
}
```

Read Tree Shaking online: https://riptutorial.com/webpack/topic/6466/tree-shaking

Chapter 8: Usage of Webpack

Examples

Usage CommonJS modules example

Create folder. Open it in command line. Run `npm install webpack -g`. Create 2 files:

cats.js:

```
var cats = ['dave', 'henry', 'martha'];
module.exports = cats;
```

app.js

```
cats = require('./cats.js');
console.log(cats);
```

Run in command line: `webpack ./app.js app.bundle.js`

Now in folder will be file `app.bundle.js`. You can include it in index.html page, open it in browser and see result in console.

But more fast way is run in command line: `node app.bundle.js` and see result immediately in console:

```
[ 'dave', 'henry', 'martha' ]
```

Usage AMD modules example

Create folder. Open it in command line. Run `npm install webpack -g`. Create 2 files:

cats.js:

```
define(function(){
    return ['dave', 'henry', 'martha'];
});
```

app.js

```
require(['./cats'],function(cats){
    console.log(cats);
})
```

Run in command line:

```
webpack ./app.js app.bundle.js
```

Now in folder will be file: `app.bundle.js`.

Create index.html file:

```
<html>
    <body>
        <script src='app.bundle.js' type="text/javascript"></script>
    </body>
</html>
```

Open it in browser and see result in console:

> ['dave', 'henry', 'martha']

Usage ES6 (Babel) modules example

as written in MDN at July 2016:

> This feature is not implemented in any browsers natively at this time. It is implemented in many transpilers, such as the Traceur Compiler, Babel or Rollup.

So here is example with Babel loader for Webpack:

Create folder. Add package.json file there:

```
{
  "devDependencies": {
    "babel-core": "^6.11.4",
    "babel-loader": "^6.2.4",
    "babel-preset-es2015": "^6.9.0",
    "webpack": "^1.13.1"
  }
}
```

Open folder in command line. Run:

```
npm install.
```

Create 2 files:

cats.js:

```
export var cats = ['dave', 'henry', 'martha'];
```

app.js:

```
import {cats} from "./cats.js";
console.log(cats);
```

For proper using of babel-loader should be added **webpack.config.js** file:

```
 module: {
  loaders: [
    {
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      loader: 'babel?presets[]=es2015'
    }
  ]
}
```

Run in command line:

> webpack ./app.js app.bundle.js

Now in folder will be file: `app.bundle.js`.

Create **index.html** file:

```
<html>
    <body>
        <script src='app.bundle.js' type="text/javascript"></script>
    </body>
</html>
```

Open it in browser and see result in console:

> ['dave', 'henry', 'martha']

Usage ES6 (Typescript) modules example

as written in [MDN][1] at July 2016:

> This feature is not implemented in any browsers natively at this time. It is implemented in many transpilers, such as the Traceur Compiler, Babel or Rollup.

So here is example with Typescript loader for Webpack:

//TODO

Create simplified version of this article: http://www.jbrantly.com/typescript-and-webpack/ without tsd and jquery.

Read Usage of Webpack online: https://riptutorial.com/webpack/topic/6001/usage-of-webpack

Credits

S. No	Chapters	Contributors
1	Getting started with webpack	BrunoLM, CodingIntrigue, Community, Everettss, Filip Dupanović, Gun, Ken Redler, m_callens, neaumusic, noah.muth, Pavlin, pongo, RamenChef, Ru Chern Chong, Toby
2	Development server: webpack-dev-server	mleko
3	DllPlugin and DllReferencePlugin	BrunoLM
4	Hot Module Replacement	BrunoLM, Kevin Law
5	Loaders	4m1r
6	Loaders & Plugins	jabacchetta, mleko
7	Tree Shaking	RationalDev
8	Usage of Webpack	Rajab Shakirov