목차

[프론트엔드 개발환경의 이해: NPM 4](#_Toc33965408)

[1. Node.js 설치 4](#_Toc33965409)

[3. 프로젝트 초기화 5](#_Toc33965410)

[3.1 INIT 5](#_Toc33965411)

[3.2 Package.json 6](#_Toc33965412)

[4. 프로젝트 명령어 6](#_Toc33965413)

[5. 패키지 설치 8](#_Toc33965414)

[5.1 CDN을 이용한 방법 8](#_Toc33965415)

[5.2 직접 다운로드하는 방법 9](#_Toc33965416)

[5.3 NPM을 이용한 방법 9](#_Toc33965417)

[5.4 유의적 버전 9](#_Toc33965418)

[5.5 버전의 범위 10](#_Toc33965419)

[6. 정리 11](#_Toc33965420)

[프론트엔드 개발환경의 이해: Babel 41](#_Toc33965421)

[1. 배경 41](#_Toc33965422)

[1.1 크로스 브라우징 41](#_Toc33965423)

[1.2 트랜스파일과 빌드 41](#_Toc33965424)

[2. 바벨의 기본 동작 41](#_Toc33965425)

[3. 플러그인 42](#_Toc33965426)

[3.1 커스텀 플러그인 43](#_Toc33965427)

[3.2 플러그인 사용하기 45](#_Toc33965428)

[4. 프리셋 47](#_Toc33965429)

[4.1 커스텀 프리셋 48](#_Toc33965430)

[4.2 프리셋 사용하기 49](#_Toc33965431)

[5. env 프리셋 설정과 폴리필 50](#_Toc33965432)

[5.1 타겟 브라우져 50](#_Toc33965433)

[5.2 폴리필 51](#_Toc33965434)

[6. 웹팩으로 통합 54](#_Toc33965435)

[7. 정리 56](#_Toc33965436)

[프론트엔드 개발환경의 이해: 린트 56](#_Toc33965437)

[1. 배경 56](#_Toc33965438)

[1.1 린트가 필요한 상황 57](#_Toc33965439)

[2. ESLint 57](#_Toc33965440)

[2.1 기본 개념 57](#_Toc33965441)

[2.2 설치 및 사용법 58](#_Toc33965442)

[2.3 규칙(Rules) 58](#_Toc33965443)

[2.4 자동으로 수정할 수 있는 규칙 59](#_Toc33965444)

[2.5 Extensible Config 60](#_Toc33965445)

[2.6 초기화 61](#_Toc33965446)

[3. Prettier 61](#_Toc33965447)

[3.1 설치 및 사용법 62](#_Toc33965448)

[3.2 포매팅(더 예쁘게) 62](#_Toc33965449)

[3.3 통합방법 63](#_Toc33965450)

[4. 자동화 66](#_Toc33965451)

[4.1 변경한 내용만 검사 66](#_Toc33965452)

[4.2 에디터 확장도구 68](#_Toc33965453)

[5. 정리 69](#_Toc33965454)

[프론트엔드 개발환경의 이해: 웹팩(심화) 70](#_Toc33965455)

[1. 웹팩 개발 서버 70](#_Toc33965456)

[1.1 배경 70](#_Toc33965457)

[1.2 설치 및 사용 70](#_Toc33965458)

[1.3 기본 설정 72](#_Toc33965459)

[2. API 연동 74](#_Toc33965460)

[2.1 목업 API 1: devServer.before 74](#_Toc33965461)

[2.2 목업 API 2: connect-api-mocker 77](#_Toc33965462)

[2.3 실제 API 연동: devServer.proxy 78](#_Toc33965463)

[3. 핫 모듈 리플레이스먼트 81](#_Toc33965464)

[3.1 배경 81](#_Toc33965465)

[3.2 설정 81](#_Toc33965466)

[3.3 핫로딩을 지원하는 로더 84](#_Toc33965467)

[4. 최적화 85](#_Toc33965468)

[4.1 production 모드 85](#_Toc33965469)

[4.2 optimazation 속성으로 최적화 86](#_Toc33965470)

[4.3 코드 스플리팅 88](#_Toc33965471)

[4.4 externals 92](#_Toc33965472)

[5. 정리 94](#_Toc33965473)

# 프론트엔드 개발환경의 이해: NPM

/d/Study/npm

$ git init

$ git remote add origin https://github.com/phjpsy/npm.git

$ git status

$ git add .

$ git commit -m "first make 20200301"

$ git push origin master

# 1. Node.js 설치

[Nodejs.org 사이트](https://nodejs.org/ko/)에서 노드 최신 버전을 설치하자.

아래 경로에 Node.js 와 NPM이 설치된다

* Node.js: C:\Program Files\nodejs

Node.js command prompt터미널을 열어 각 명령어 확인.

$ node

> 1 + 2

3

.exit 실행하거나 ctrl + c를 연속 두 번 입력하면 REPL 프로그램에서 빠져 나올 수 있다.

$ node --version

v13.1.0

NPM(Node Package Manage) 명령어도 확인해 보자.

$ npm

Usage: npm <command>

$ npm --version

6.12.1

# 3. 프로젝트 초기화

NPM은 자바스크립트 기반 프로젝트의 빌드 도구인 셈이다. NPM을 이용해 프론트엔드 개발 프로젝트를 세팅해 보자.

## 3.1 INIT (프로젝트를 생성)

$ npm init

package name:

version:

description:

entry point:

test command:

git repository:

keywords:

author:

license:

모든 질문에 답하면 명령어를 실행한 폴더에 pacakge.json 파일이 생성된다. 모두 기본값을 사용할 것이라면 npm init -y 명령어로 질문을 스킵하고 package.json 파일을 생성할 수 있다.

## 3.2 Package.json

Node.js는 package.json 파일에 프로젝트의 모든 정보를 기록한다.

* name: 프로젝트 이름
* version: 프로젝트 버전 정보
* description: 프로젝트 설명
* main: 노드 어플리케이션일 경우 진입점 경로. 프론트엔드 프로젝트일 경우 사용하지 않는다.
* scripts: 프로젝트 명령어를 등록할 수 있다.초기화시 test 명령어가 샘플로 등록되어 있다
* author: 프로그램 작성자
* license: 라이센스

# 4. 프로젝트 명령어

생성한 프로젝트는 package.json에 등록한 스크립트(scripts)를 이용해 실행한다. 어플리케이션 빌드, 테스트, 배포, 실행 따위의 명령어를 등록하는데 실행하는 방법은 다음과 같다.

$ npm test

package.json:

{… "scripts": { "test": "node" }, …}

NPM에서 사용할 수 있는 명령어는 몇개나 될까? NPM 사용법을 확인해 보자.

D:\Study\npm\test>npm

Usage: npm <command>

where <command> is one of:

access, adduser, audit, bin, bugs, c, cache, ci, cit,

clean-install, clean-install-test, completion, config,

create, ddp, dedupe, deprecate, dist-tag, docs, doctor,

edit, explore, get, help, help-search, hook, i, init,

install, install-ci-test, install-test, it, link, list, ln,

login, logout, ls, org, outdated, owner, pack, ping, prefix,

profile, prune, publish, rb, rebuild, repo, restart, root,

run, run-script, s, se, search, set, shrinkwrap, star,

stars, start, stop, t, team, test, token, tst, un,

uninstall, unpublish, unstar, up, update, v, version, view,

whoami

무척 많다. 보통 사용하는 것들이 start, test, install, uninstall이다.

* start: 어플리케이션 실행
* test: 테스트
* install: 패키지 설치
* uninstall: 패키지 삭제

명령어를 추가할 수도 있다. 가령 빌드를 위한 build 스크립트는 package.json의 scripts 부분에 build 키를 추가하고 쉘 스크립트를 문자열로 등록하면 된다.

package.json:

"scripts": { "build": "여기에 빌드 스크립트를 등록한다" }}

커스텀으로 등록한 명령어는 아래 형식으로 run을 추가해서 실행한다.

$ npm run build

프론트엔드 개발 환경을 구축하기 위해 추가할 스크립트는 build, lint 정도가 있겠다.

* build: 소스 빌드
* lint: 소스 컨벤션 검사

# 5. 패키지 설치

## 5.1 CDN을 이용한 방법

외부 라이브러리를 가져다 쓰는 것은 무척 자연스러운 일이다. 간단한 방법은 CDN(컨텐츠 전송 네트워크)으로 제공하는 라이브러리를 직접 가져 오는 방식이다. 리액트의 주소를 html에서 로딩한다.

<script src="<https://unpkg.com/react@16/umd/react.development.js>"></script>

CDN 서버 장애로 인해 외부 라이브러리를 사용할 수 없다면 어떻게 될까? 아무리 우리 어플리케이션 서버가 정상이더라도 필수 라이브러리를 가져오지 못한다면 웹 어플리케이션은 정상적으로 동작하지 않을 것이다.

## 5.2 직접 다운로드하는 방법

라이브러리 코드를 우리 프로젝트 폴더에 다운받아 놓는건 어떨까? CDN을 사용하지 않기 때문에 장애와 독립적으로 웹 어플리케이션을 제공할 수 있을 것같다.

하지만 이런 상황도 있다. 라이브러리는 계속해서 업데이트 될 것이고 우리 프로젝트에서도 최신 버전으로 교체해야 한다. 매번 직접 다운로드하는 것은 매우 귀찮은 일이 될 것이다. 버전에 따라 하위 호환성 여부까지 확인하려면 실수할 여지가 많다.

라이브러리를 어느 한 곳에서 업데이트하고 하위 호환되는 안전한 버전만 다운받아 사용할 수 있다면 어떨까?

## 5.3 NPM을 이용한 방법

NPM은 이러한 방식으로 패키지를 관리한다. npm install 명령어로 외부 패키지를 우리 프로젝트 폴더에 다운로드 해보자.

$ npm install react

최신 버전의 react를 [NPM 저장소](https://www.npmjs.com/)에서 찾아 우리 프로젝트로 다운로드 하는 명령어다.  
package.json에는 설치한 패키지 정보를 기록한다.

package.json:

{

"dependencies": {

"react": "^16.12.0"

}

}

버전 16.12.0을 설치했다는 의미다.

## 5.4 유의적 버전

“^16.12.0” 이 표기는 무슨 뜻일까?

위 질문에 답하기 전에 버전 관리에 대해서 생각해 보자. 만약 프로젝트에서 사용하는 패키지의 버전을 엄격하게 제한한다면 어떨까? 프로젝트를 버전업 하는데 꽤 힘들 수 있다. 사용하는 패키지를 전부 버전업해야 하기 때문이다. 어쩌면 우리 프로젝트는 현재 버전에 갖혀 버릴지도 모른다.

그럼 프로젝트에서 사용하는 패키지 버전을 느슨하게 풀어 놓으면 문제가 해결될까? 오히려 여러 버전별로 코드를 관리해야하는 혼란스러움을 겪게될 수 있다.

버전 번호를 관리하기 위한 규칙이 필요한데 이 체계를 **“유의적 버전”**이라고 한다. NPM은 이 [유의적 버전(Sementic Version)](https://semver.org/lang/ko/)을 따르는 전제 아래 패키지 버전을 관리한다.

유의적 버전은 주(Major), 부(Minor), 수(Patch) 세 가지 숫자를 조합해서 버전을 관리한다. 위에 설치한 react의 버전은 v16.12.0인데 주 버전이 16, 부 버전이 12, 수 버전이 0인 셈이다.

각 버전을 변경하는 기준은 다음과 같다

* 주 버전(Major Version): 기존 버전과 호환되지 않게 변경한 경우
* 부 버전(Minor version): 기존 버전과 호환되면서 기능이 추가된 경우
* 수 버전(Patch version): 기존 버전과 호환되면서 버그를 수정한 경우

## 5.5 버전의 범위

NPM이 버전을 관리하는 방식은 유의적 버전 명세 뿐만아니라 버전의 범위를 자신만의 규칙으로 관리한다. 가장 단순한 것이 특정 버전을 사용하는 경우다.

1.2.3

특정 버전보다 높거나 낮을 경우는 다음과 같이 명시한다.

>1.2.3

>=1.2.3

<1.2.3

<=1.2.3

마지막으로 틸드(~)와 캐럿(^)을 이용해 범위를 명시한다.

~1.2.3

^1.2.3

[**틸트(~)**](https://docs.npmjs.com/misc/semver#tilde-ranges-123-12-1)는 마이너 버전이 명시되어 있으면 패치버전을 변경한다. 예를 들어 ~1.2.3 표기는 1.2.3 부터 1.3.0 미만 까지를 포함한다. 마이너 버전이 없으면 마이너 버전을 갱신한다. ~0 표기는 0.0.0부터 1.0.0 미만 까지를 포함한다.

[**캐럿(^)**](https://docs.npmjs.com/misc/semver#caret-ranges-123-025-004)은 정식버전에서 마이너와 패치 버전을 변경한다. 예를 들어 ^1.2.3 표기는 1.2.3부터 2.0.0 미만 까지를 포함한다. 정식버전 미만인 0.x 버전은 패치만 갱신한다. ^0 표기는 0.0.0부터 0.1.0 미만 까지를 포함한다.

보통 라이브러리 정식 릴리즈 전에는 패키지 버전이 수시로 변한다. 0.1에서 0.2로 부버전이 변하더라도 하위 호환성을 지키지 않고 배포하는 경우가 빈번하다. ~0로 버전 범위를 표기한다면 0.0.0부터 1.0.0미만까지 사용하기 때문에 하위 호완성을 지키지 못하는 0.2로도 업데이트 되어버리는 문제가 생길수 있다.

반면 캐럿을 사용해 ^0.0으로 표기한다면 0.0.0부터 0.1.0 미만 내에서만 버전을 사용하도록 제한한다. 따라서 하위 호완성을 유지할 수 있다. (자세한 내용은 [여기](https://blog.outsider.ne.kr/1041)를 참고)

NPM으로 패키지를 설치하면 package.json에 설치한 버전을 기록하는데 캐럿 방식을 이용한다. 초기에는 버전 범위에 틸트를 사용하다가 캐럿을 도입해서 기본 동작으로 사용했다. 그래서 우리가 설치한 react는 ^16.12.0 표기로 버전 범위를 기록한 것이다.

# 6. 정리

Node.js 기술을 기반으로 하는 프로트엔드 개발 환경 구축을 위해 Node.js와 NPM을 설치했다.

npm init 명령어를 사용하면 package.json에 정보를 기록하고 프로젝트를 초기화 한다.

NPM이 제공하는 기본 명령어와 커스텀 명령어 추가방법을 알아보았다.

npm install로 외부 패키지를 다운로드 할 수 있고, 버전을 관리하는 방식에 대해 살펴 보았다.

프론트엔드 개발환경의 이해: 웹팩(기본)

2019년 12월 10일

1. 배경

먼저 모듈에 대해 이야기 해보자. 문법 수준에서 모듈을 지원하기 시작한 것은 ES2015부터다. import/export 구문이 없었던 모듈 이전 상황을 살펴보는 것이 웹팩 등장 배경을 설명하는데 수월할 것 같다.

아래 덧셈 함수를 보자.

math.js:

function sum(a, b) { return a + b; } // 전역 공간에 sum이 노출

app.js:

sum(1, 2); // 3

위 코드는 모두 하나의 HTML 파일 안에서 로딩해야만 실행된다. math.js가 로딩되면 app.js는 이름 공간에서 ‘sum’을 찾은 뒤 이 함수를 실행한다. 문제는 ‘sum’이 전역 공간에 노출된다는 것. 다른 파일에서도 ‘sum’이란 이름을 사용한다면 충돌한다.

1.1 IIFE 방식의 모듈

이러한 문제를 예방하기 위해 스코프를 사용한다. 함수 스코프를 만들어 외부에서 안으로 접근하지 못하도록 공간을 격리하는 것이다. 스코프 안에서는 자신만의 이름 공간이 존재하므로 스코프 외부와 이름 충돌을 막을 수 있다.

math.js:

var math = math || {}; // math 네임스페이스

(function() {

function sum(a, b) { return a + b; }

math.sum = sum; // 네이스페이스에 추가

})();

같은 코드를 즉시실행함수로 감싸서 다른 파일에서 이 안으로 접근할 수가 없다. 심지어 같은 파일일지라도 말이다. 자바스크립트 함수 스코프의 특징이다. ‘sum’이란 이름은 즉시실행함수 안에 감추어졌기 때문에 외부에서는 같은 이름을 사용해도 괜찮다. 전역에 등록한 ‘math’라는 이름 공간만 잘 활용하면 된다.

1.2 다양한 모듈 스펙

이러한 방식으로 자바스크립트 모듈을 구현하는 대표적인 명세가 AMD와 CommonJS다.

[CommonJS](http://www.commonjs.org/)는 자바스크립트를 사용하는 모든 환경에서 모듈을 하는 것이 목표다. exports 키워드로 모듈을 만들고 require() 함수로 불러 들이는 방식이다. 대표적으로 서버 사이드 플래폼인 Nodejs에서 이를 사용한다.

math.js:

exports function sum(a, b) { return a + b; }

app.js:

const math = require('./math.js');

math.sum(1, 2); // 3

[AMD](https://github.com/amdjs/amdjs-api/wiki/AMD)(Asynchronous Module Definition)는 비동기로 로딩되는 환경에서 모듈을 사용하는 것이 목표다. 주로 브라우져 환경이다.

[UMD](https://github.com/umdjs/umd)(Universal Module Definition)는 AMD기반으로 CommonJS 방식까지 지원하는 통합 형태다.

이렇게 각 커뮤니티에서 각자의 스펙을 제안하다가 [ES2015에서 표준 모듈 시스템](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import)을 내 놓았다. 지금은 바벨과 웹팩을 이용해 모듈 시스템을 사용하는 것이 일반적이다. ES2015 모듈 시스템의 모습을 살펴보자.

math.js:

export function sum(a, b) { return a + b; }

app.js:

import \* as math from './math.js';

math.sum(1, 2); // 3

export 구문으로 모듈을 만들고 import 구문으로 가져올 수 있다.

1.3 브라우져의 모듈 지원

안타깝게도 모든 브라우져에서 모듈 시스템을 지원하지는 않는다. 인터넷 익스플로러를 포함한 몇 몇 브라우져에서는 여전히 모듈을 사용하지 못한다. 가장 많이 사용하는 크롬 브라우져만 잠시 살펴보자. ([버전 61부터 모듈시스템을 지원](https://developers.google.com/web/updates/2017/09/nic61#modules) 한다)

index.html:

<script type="module" src="app.js"></script>

<script> 태그로 로딩할 때 type="text/javascript" 대신 type="module"을 사용한다. app.js는 모듈을 사용할 수 있다.

그러나 브라우져에 무관하게 모듈을 사용하고 싶은데…… 이제야 웹팩이 나올 차례다.

2. 엔트리/아웃풋

[웹팩](https://webpack.js.org/)은 여러개 파일을 하나의 파일로 합쳐주는 번들러(bundler)다. 하나의 시작점(entry point)으로부터 의존적인 모듈을 전부 찾아내서 하나의 결과물을 만들어 낸다. app.js부터 시작해 math.js 파일을 찾은 뒤 하나의 파일로 만드는 방식이다.

간단히 웹팩으로 번들링 작업을 해보자.

번들 작업을 하는 [webpack](https://github.com/webpack/webpack) 패키지와 웹팩 터미널 도구인 [webpack-cli](https://github.com/webpack/webpack-cli)를 설치한다.

$ npm install -D webpack webpack-cli

설치 완료하면 node\_modules/.bin 폴더에 실행 가능한 명령어가 몇 개 생긴다. webpack과 webpack-cli가 있는데 둘 중 하나를 실행하면 된다. --help 옵션으로 사용 방법을 확인해 보자.

$ node\_modules/.bin/webpack --help

--mode Enable production optimizations or development hints.

[선택: "development", "production", "none"]

--entry The entry point(s) of the compilation. [문자열]

--output, -o The output path and file for compilation assets

–mode, –entry, –output 세 개 옵션만 사용하면 우선 번들링 할 수 있다.

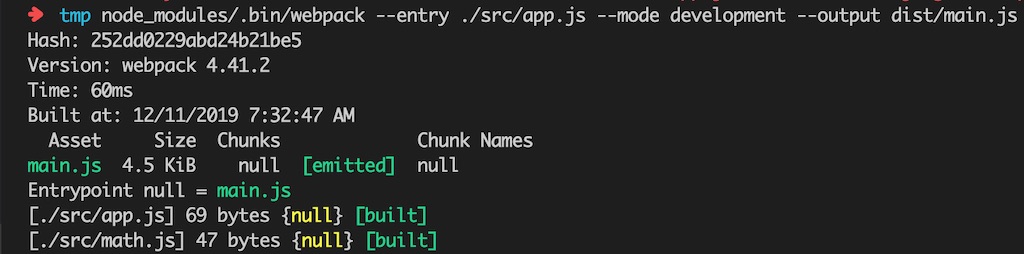
$ node\_modules/.bin/webpack --mode development --entry ./src/app.js --output dist/main.js

--mode는 웹팩 실행 모드는 의미하는데 개발 버전인 development를 지정한다

--entry는 시작점 경로를 지정하는 옵션이다

--output은 번들링 결과물을 위치할 경로다

위 명령어를 실행하면 dist/main.js에 번들된 결과가 저장된다.



이 코드를 index.html에 로딩하면 번들링 전과 똑같은 결과를 만든다.

index.html:

<script src="dist/main.js"></script>

옵션 중 --config 항목을 보자.

$ node\_modules/.bin/webpack --help

--config Path to the config file

[문자열] [기본: webpack.config.js or webpackfile.js]

이 옵션은 웹팩 설정파일의 경로를 지정할 수 있는데 기본 파일명이 webpack.config.js 혹은 webpackfile.js다. webpack.config.js 파일을 만들어 방금 터미널에서 사용한 옵션을 코드로 구성해 보자.

webpack.config.js:

const path = require('path');

module.exports = {

mode: 'development',

entry: {

main: './src/app.js'

},

output: {

filename: '[name].js',

path: path.resolve('./dist'),

},

}

터미널에서 사용한 옵션인 mode, entry, ouput을 설정한다.

mode는 ‘development’ 문자열을 사용했다.

entry는 어플리케이션 진입점인 src/app.js로 설정한다.

ouput에 설정한 ‘[name]’은 entry에 추가한 main이 문자열로 들어오는 방식이다.

output.path는 절대 경로를 사용하기 때문에 path 모듈의 resolve() 함수를 사용해서 계산했다. (path는 노드 코어 모듈 중 하나로 경로를 처리하는 기능을 제공한다)

웹팩 실행을 위한 NPM 커스텀 명령어를 추가한다.

package.json:

{

"scripts": {

"build": "./node\_modules/.bin/webpack"

}

}

모든 옵션을 웹팩 설정 파일로 옮겼기 때문에 단순히 webpack 명령어만 실행한다. 이제부터는 npm run build로 웹팩 작업을 지시할 수 있다.

3. 로더

3.1 로더의 역할

웹팩은 모든 파일을 모듈로 바라본다. 자바스크립트로 만든 모듈 뿐만아니라 스타일시트, 이미지, 폰트까지도 전부 모듈로 보기 때문에 import 구문을 사용하면 자바스크립트 코드 안으로 가져올수 있다.

이것이 가능한 이유는 웹팩의 로더 덕분이다. 로더는 타입스크립트 같은 다른 언어를 자바스크립트 문법으로 변환해 주거나 이미지를 data URL 형식의 문자열로 변환한다. 뿐만아니라 CSS 파일을 자바스크립트에서 직접 로딩할수 있도록 해준다.

3.2 커스텀 로더 만들기

로더를 사용하기 전에 동작 원리를 이해하기 위해 로더를 직접 만들어 보자.

myloader.js:

module.exports = function myloader (content) {

console.log('myloader가 동작함');

return content;

};

함수로 만들수 있는데 로더가 읽은 파일의 내용이 함수 인자 content로 전달된다. 로더가 동작하는지 확인하는 용도로 로그만 찍고 곧장 content를 돌려 준다.

로더를 사용하려면 웹팩 설정파일의 module 객체에 추가한다.

webpack.config.js:

module: {

rules: [{

test: /\.js$/, // .js 확장자로 끝나는 모든 파일

use: [path.resolve('./myloader.js')] // 방금 만든 로더를 적용한다

}],

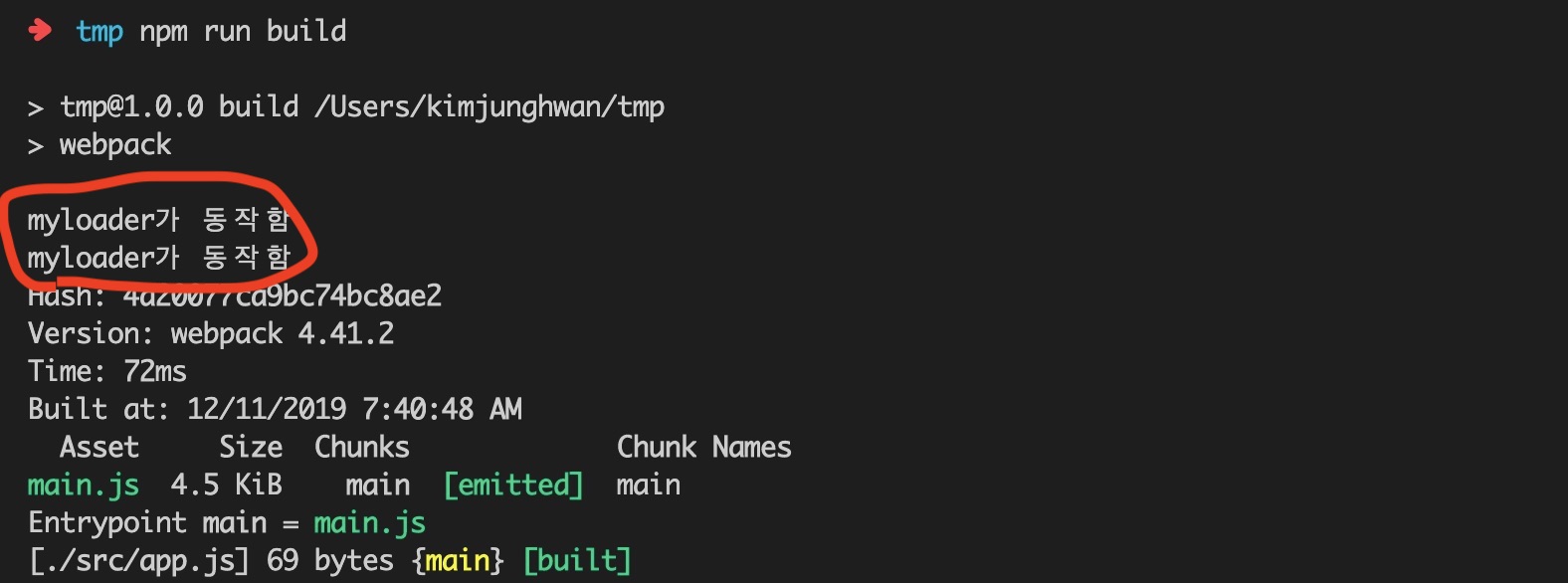
}

module.rules 배열에 모듈을 추가하는데 test와 use로 구성된 객체를 전달한다.

test에는 로딩에 적용할 파일을 지정한다. 파일명 뿐만아니라 파일 패턴을 정규표현식으로 지정할수 있는데 위 코드는 .js 확장자를 갖는 모든 파일을 처리하겠다는 의미다.

use에는 이 패턴에 해당하는 파일에 적용할 로더를 설정하는 부분이다. 방금 만든 myloader 함수의 경로를 지정한다.

이제 npm run build로 웹팩을 실행해 보자.



터미널에 ‘myloader가 동작함’ 문자열이 찍힌다. myloader() 함수가 동작한 것이다.

빌드결과를 살펴보면 이전과 동일하다. 로더가 뭔가를 처리하기 위해서 간단한 변환 작업을 추가해 보자. 소스에 있는 모든 console.log() 함수를 alert() 함수로 변경하도록 말이다.

myloader.js:

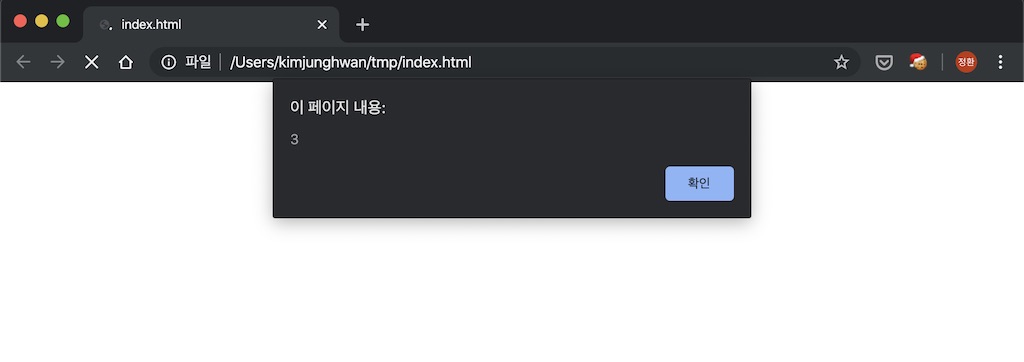
module.exports = function myloader (content) {

console.log('myloader가 동작함')

return content.replace('console.log(', 'alert('); // console.log( -> alert( 로 치환

};

빌드후 확인하면 다음과 같이 console.log() 함수가 alert() 함수로 변경되었다.



4. 자주 사용하는 로더

로더의 동작 원리를 살펴 보았으니 이번에는 몇몇 자주 사용하는 로더를 소개하겠다.

4.1 css-loader

웹팩은 모든것을 모듈로 바라보기 때문에 자바스크립트 뿐만 아니라 스타일시트로 import 구문으로 불러 올수 있다.

app.js:

import './style.css'

style.css:

body {

background-color: green;

}

CSS 파일을 자바스크립트에서 불러와 사용하려면 CSS를 모듈로 변환하는 작업이 필요하다. [css-loader](https://github.com/webpack-contrib/css-loader)가 그러한 역할을 하는데 우리 코드에서 CSS 파일을 모듈처럼 불러와 사용할 수 있게끔 해준다.

먼저 로더를 설치 하자.

$ npm install -D css-loader

웹팩 설정에 로더를 추가한다.

webpack.config.js:

module.exports = {

module: {

rules: [{

test: /\.css$/, // .css 확장자로 끝나는 모든 파일

use: ['css-loader'], // css-loader를 적용한다

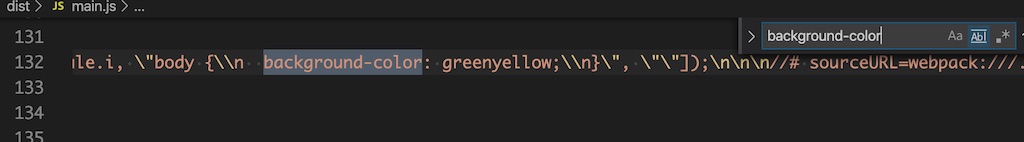
}]

}

}

웹팩은 엔트리 포인트부터 시작해서 모듈을 검색하다가 CSS 파일을 찾으면 css-loader로 처리할 것이다. use.loader에 로더 경로를 설정하는 대신 배열에 로더 이름을 문자열로 전달해도 된다.

빌드 한 결과 CSS코드가 자바스크립트로 변환된 것을 확인할 수 있다.



4.2 style-loader

모듈로 변경된 스타일 시트는 돔에 추가되어야만 브라우져가 해석할 수 있다. css-loader로 처리하면 자바스크립트 코드로만 변경되었을 뿐 돔에 적용되지 않았기 때문에 스트일시트가 적용되지 않았다. [style-loader](https://github.com/webpack-contrib/style-loader)는 자바스크립트로 변경된 스타일시트를 동적으로 돔에 추가하는 로더이다. CSS를 번들링하기 위해서는 css-loader와 style-loader를 함께 사용한다.

먼저 스타일 로더를 다운로드 한다.

$ npm install -D style-loader

그리고 웹팩 설정에 로더를 추가한다.

package.json:

module.exports = {

module: {

rules: [{

test: /\.css$/,

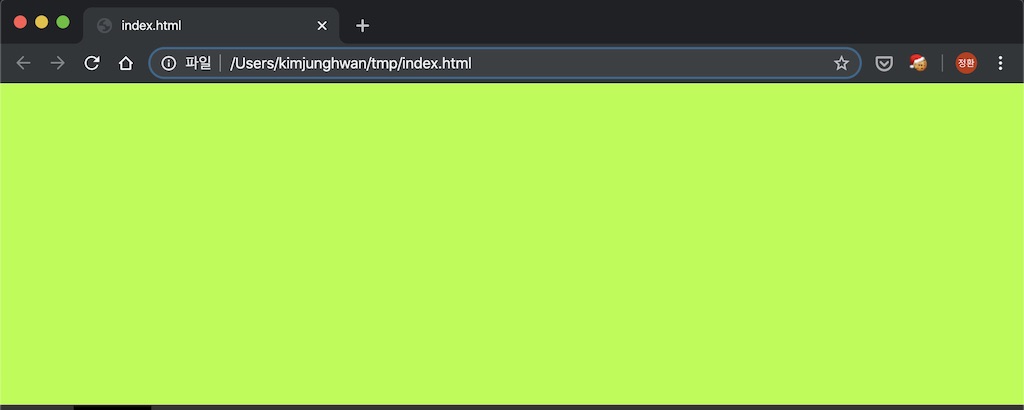
use: ['style-loader', 'css-loader'], // style-loader를 앞에 추가한다

}]

}

}

배열로 설정하면 뒤에서부터 앞으로 로더가 동작한다.  
위 설정은 모든 .css 확장자로 끝나는 모듈을 읽어 들여 css-loader를 적용하고 그 다음 style-loader를 적용한다.



4.3 file-loader

CSS 뿐만 아니라 소스코드에서 사용하는 모든 파일을 모듈로 사용하게끔 할 수 있다. 파일을 모듈 형태로 지원하고 웹팩 아웃풋에 파일을 옮겨주는 것이 [file-loader](https://github.com/webpack-contrib/file-loader)가 하는 일이다. 가령 CSS에서 url() 함수에 이미지 파일 경로를 지정할 수 있는데 웹팩은 file-loader를 이용해서 이 파일을 처리한다.

style.css:

body {

background-image: url(bg.png);

}

배경 이미지를 bg.png 파일로 지정했다.

웹팩은 엔트리 포인트인 app.js가 로딩하는 style.css 파일을 읽을 것이다. 그리고 이 스타일시트는 url() 함수로 bg.png를 사용하는데 이때 로더를 동작시킨다.

webpack.config.js:

module.exports = {

module: {

rules: [{

test: /\.png$/, // .png 확장자로 마치는 모든 파일

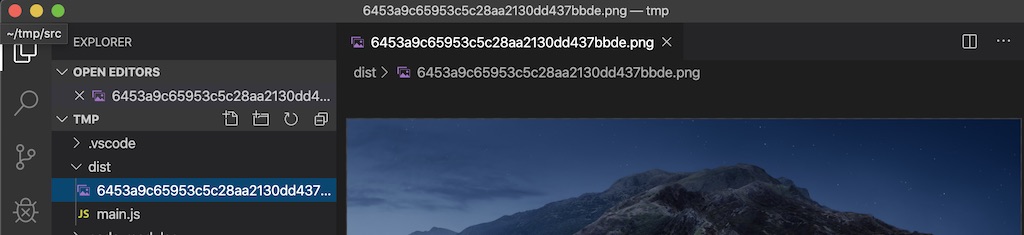
loader: 'file-loader', // 파일 로더를 적용한다

}]

}

}

웹팩이 .png 파일을 발견하면 file-loader를 실행할 것이다. 로더가 동작하고 나면 아웃풋에 설정한 경로로 이미지 파일을 복사된다. 아래 그림처럼 파일명이 해쉬코드로 변경 되었다. 캐쉬 갱신을 위한 처리로 보인다.



하지만 이대로 index.html 파일을 브라우져에 로딩하면 이미지를 제대로 로딩하지 못할 것이다. CSS를 로딩하면 background-image: url(bg.png) 코드에 의해 동일 폴더에서 이미지를 찾으려고 시도할 것이다. 그러나 웹팩으로 빌드한 이미지 파일은 output인 dist 폴더 아래로 이동했기 때문에 이미지 로딩에 실패할 것이다.

file-loader 옵션을 조정해서 경로를 바로 잡아 주어야 한다.

module.exports = {

module: {

rules: [{

test: /\.png$/, // .png 확장자로 마치는 모든 파일

loader: 'file-loader',

options: {

publicPath: './dist/', // prefix를 아웃풋 경로로 지정

name: '[name].[ext]?[hash]', // 파일명 형식

}

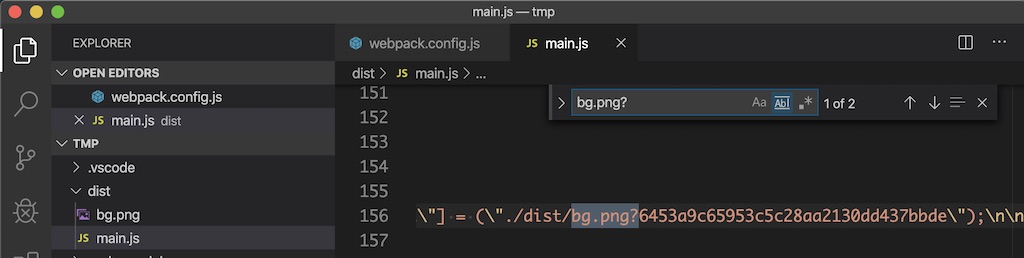
}]

}

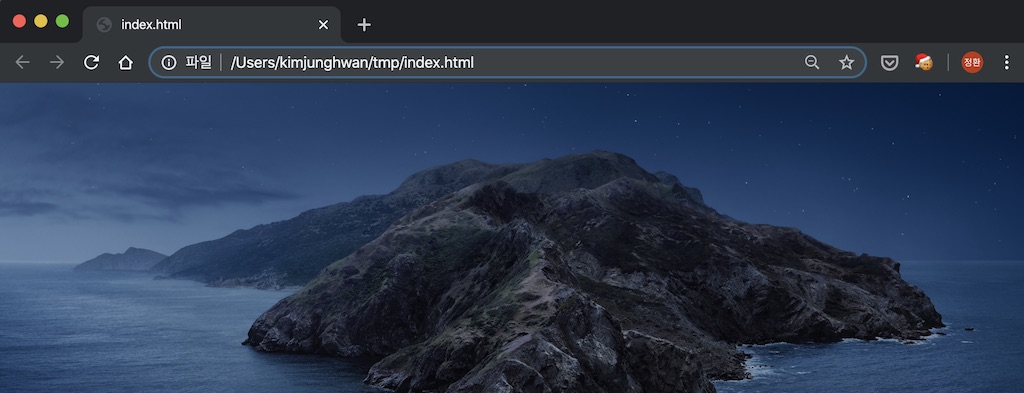
}

publicPath 옵션은 file-loader가 처리하는 파일을 모듈로 사용할 때 경로 앞에 추가되는 문자열이다. output에 설정한 ‘dist’ 폴더에 이미지 파일을 옮길 것이므로 publicPath 값을 이것으로로 지정했다. 파일을 사용하는 측에서는 ‘bg.png’를 ‘dist/bg.png’로 변경하여 사용할 것이다.

또한 name 옵션을 사용했는데 이것은 로더가 파일을 아웃풋에 복사할때 사용하는 파일 이름이다. 기본적으로 설정된 해쉬값을 쿼리스트링으로 옮겨서 ‘bg.png?6453a9c65953c5c28aa2130dd437bbde’ 형식으로 파일을 요청하도록 변경했다.



이렇게 스타일시트에서 불러온 파일이 동작한다.



4.4 url-loader

사용하는 이미지 갯수가 많다면 네트웍 리소스를 사용하는 부담이 있고 사이트 성능에 영향을 줄 수도 있다. 만약 한 페이지에서 작은 이미지를 여러개 사용한다면 [Data URI Scheme](https://en.wikipedia.org/wiki/Data_URI_scheme)을 이용하는 방법이 더 낫다. 이미지를 Base64로 인코딩하여 문자열 형태로 소스코드에 넣는 형식이다. [url-loader](https://github.com/webpack-contrib/url-loader)는 이러한 처리를 자동화해주는 녀석이다.

먼저 로더를 설치한다.

$ npm install -D url-loader

그리고 웹팩 설정을 추가한다. webpack.config.js:

{

test: /\.png$/,

use: {

loader: 'url-loader', // url 로더를 설정한다

options: {

publicPath: './dist/', // file-loader와 동일

name: '[name].[ext]?[hash]', // file-loader와 동일

limit: 5000 // 5kb 미만 파일만 data url로 처리

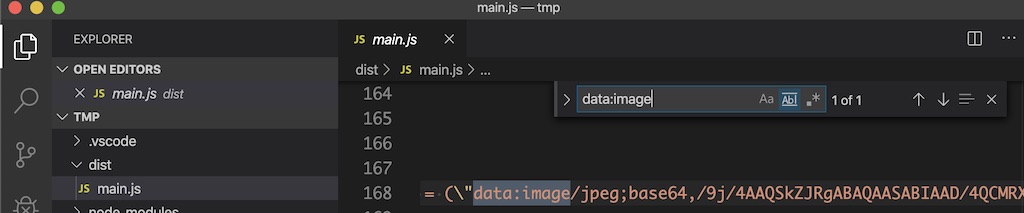
}

}

}

file-loader와 옵션 설정이 거의 비슷하고 마지막 limit 속성만 추가했다. 모듈로 사용한 파일중 크기가 5kb 미만인 파일만 url-loader를 적용하는 설정이다. 만약 이보다 크면 file-loader가 처리하는데 옵션 중 [fallback](https://github.com/webpack-contrib/url-loader#options) 기본값이 file-loader이기 때문이다.

빌드 결과를 보면 small.png 파일이 문자열로 변경되어 있는 것을 확인 할 수 있다. 반면 5kb 이상인 bg.png는 여전히 파일로 존재한다.



브라우저에서도 확인하면 스타일스트에 small.png가 Data url형태로 변환되어 있다.



아이콘처럼 용량이 작거나 사용 빈도가 높은 이미지는 파일을 그대로 사용하기 보다는 Data URI Scheeme을 적용하기 위해 url-loader를 사용하면 좋겠다.

5. 플러그인

5.1 플러그인의 역할

웹팩에서 알아야 할 마지막 기본 개념이 플러그인이다. 로더가 파일 단위로 처리하는 반면 플러그인은 번들된 결과물을 처리한다. 번들된 자바스크립트를 난독화 한다거나 특정 텍스트를 추출하는 용도로 사용한다.

이것도 사용하기에 앞서 동작 원리를 이해하기 위해 플러그인을 직접 만들어 보자.

5.2 커스텀 플러그인 만들기

웹팩 문서의 [Writing a plugin](https://webpack.js.org/contribute/writing-a-plugin/)을 보면 클래스로 플러그인을 정의 하도록 한다. [헬로월드 코드](https://webpack.js.org/contribute/writing-a-plugin/#basic-plugin-architecture)를 가져다 그대로 실행 붙여보자.

myplugin.js:

class MyPlugin {

apply(compiler) {

compiler.hooks.done.tap('My Plugin', stats => {

console.log('MyPlugin: done');

})

}

}

module.exports = MyPlugin;

로더와 다르게 플러그인은 클래스로 제작한다. apply 함수를 구현하면 되는데 이 코드에서는 인자로 받은 compiler의 tap 함수를 사용했다. 플러그인 작업이 완료되는(done) 시점에 로그를 찍는 코드인것 같다.

플러그인을 웹팩 설정에 추가한다.

webpack.config.js:

const MyPlugin = require('./myplugin');

module.exports = {

plugins: [

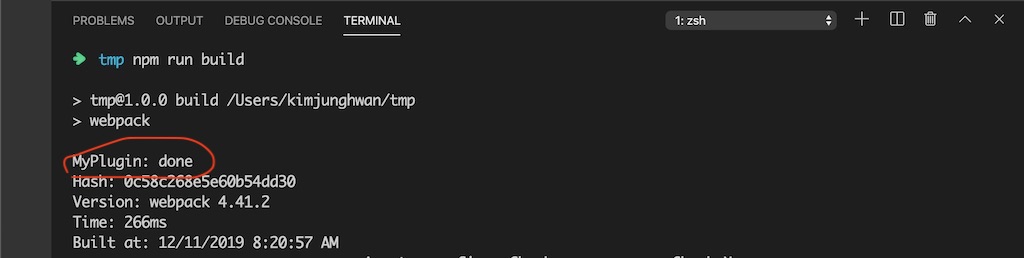
new MyPlugin(),

]

}

웹팩 설정 객체의 plugins 배열에 설정한다. 클래스로 제공되는 플러그인의 생성자 함수를 실행해서 넘기는 방식이다.

웹팩으로 빌드해 보자.



로그가 찍힌걸 보니 플러그인이 동작했다.

그런데 파일이 여러개인데 로그는 한 번만 찍혔다. 모듈은 설정한 파일 하나 혹은 여러 개에 대해 동작하지만, 플러그인은 그것을 하나로 번들링한 결과물을 대상으로 한다. 우리 예제에서는 main.js로 결과물이 하나이기 때문에 플러그인이 한번만 동작한 것이다.

그러면 어떻게 번들 결과에 접근할 수 있을까? 웹팩 내장 플러그인 [BannerPlugin 코드](https://github.com/lcxfs1991/banner-webpack-plugin/blob/master/index.js)를 참고하자.

myplugin.js:

class MyPlugin {

apply(compiler) {

compiler.hooks.done.tap('My Plugin', stats => {

console.log('MyPlugin: done');

})

compiler.plugin('emit', (compilation, callback) => { // compiler.plugin() 함수로 후처리한다

const source = compilation.assets['main.js'].source();

console.log(source);

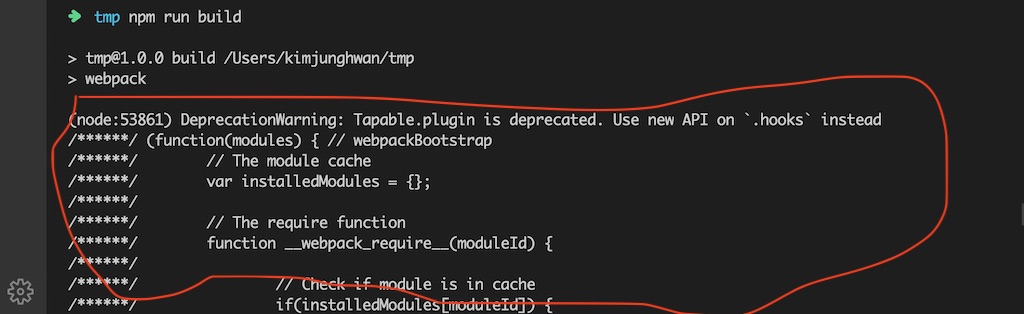
callback();

})

}

}

compiler.plugin() 함수의 두번재 인자 콜백함수는 emit 이벤트가 발생하면 실행되는 녀석인 모양이다. 번들된 결과가 compilation 객체에 들어 있는데 compilation.assets[‘main.js’].source() 함수로 접근할 수 있다. 실행하면 터미널에 번들링된 결과물을 확인할 수 있다.



이걸 이용해서 번들 결과 상단에 아래와 같은 배너를 추가하는 플러그인으로 만들어 보자.

myplugin.js:

class MyPlugin {

apply(compiler) {

compiler.plugin('emit', (compilation, callback) => {

const source = compilation.assets['main.js'].source();

compilation.assets['main.js'].source = () => {

const banner = [

'/\*\*',

' \* 이것은 BannerPlugin이 처리한 결과입니다.',

' \* Build Date: 2019-10-10',

' \*/'

].join('\n');

return banner + '\n\n' + source;

}

callback();

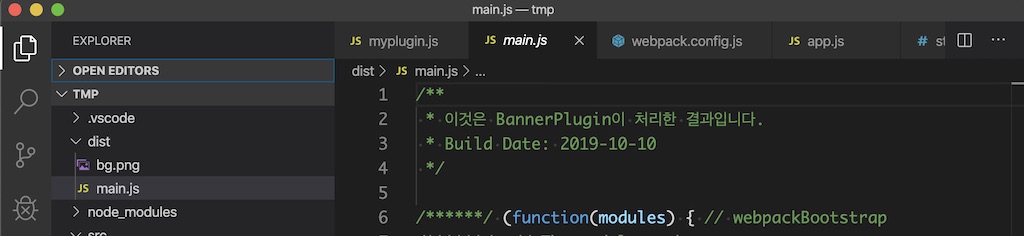
})

}

}

번들 소스를 얻어오는 함수 source()를 재정의 했다. 배너 문자열과 기존 소스 코드를 합친 문자열을 반환하도록 말이다.

빌드하고 결과물을 확인해 보면 다음과 같다.



6. 자주 사용하는 플러그인

개발하면서 플러그인을 직접 작성할 일은 거의 없었다. 웹팩에서 직접 제공하는 플러그인을 사용하거나 써드파티 라이브러리를 찾아 사용하는데 자주 사용하는 플러그인에 대해 알아보자.

6.1 BannerPlugin

MyPlugin와 비슷한 것이 [BannerPlugin](https://webpack.js.org/plugins/banner-plugin/)이다. 결과물에 빌드 정보나 커밋 버전같은 걸 추가할 수 있다.

webpack.config.js:

const webpack = require('webpack');

module.exports = {

plugins: [

new webpack.BannerPlugin({

banner: '이것은 배너 입니다',

})

]

생성자 함수에 전달하는 옵션 객체의 banner 속성에 문자열을 전달한다. 웹팩 컴파일 타임에 얻을 수 있는 정보, 가령 빌드 시간이나 커밋정보를 전달하기위해 함수로 전달할 수도 있다.

new webpack.BannerPlugin({

banner: () => `빌드 날짜: ${new Date().toLocaleString()}`

})

배너 정보가 많다면 별로 파일로 분리하자.

const banner = require('./banner.js');

new webpack.BannerPlugin(banner);

빌드 날짜 외에서 커밋 해쉬와 빌드한 유저 정보까지 추가해 보자.

banner.js:

const childProcess = require('child\_process');

module.exports = function banner() {

const commit = childProcess.execSync('git rev-parse --short HEAD')

const user = childProcess.execSync('git config user.name')

const date = new Date().toLocaleString();

return (

`commitVersion: ${commit}` +

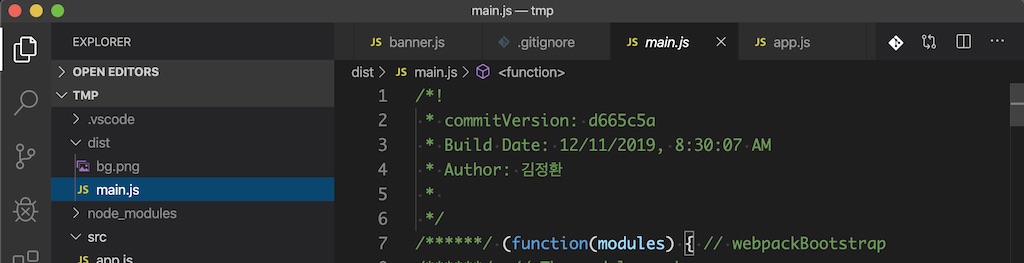
`Build Date: ${date}\n` +

`Author: ${user}`

);

}

빌드한뒤 플러그인이 처리한 결과는 다음과 같다.



6.2 DefinePlugin

어플리케이션은 개발환경과 운영환경으로 나눠서 운영한다. 가령 환경에 따라 API 서버 주소가 다를 수 있다. 같은 소스 코드를 두 환경에 배포하기 위해서는 이러한 환경 의존적인 정보를 소스가 아닌 곳에서 관리하는 것이 좋다. 배포할 때마다 코드를 수정하는 것은 곤란하기 때문이다.

웹팩은 이러한 환경 정보를 제공하기 위해 [DefinePlugin](https://webpack.js.org/plugins/define-plugin/)을 제공한다.

webpack.config.js

const webpack = require('webpack');

export default {

plugins: [

new webpack.DefinePlugin({}),

]

}

빈 객체를 전달해도 기본적으로 넣어주는 값이 있다. 노드 환경정보인 process.env.NODE\_ENV 인데 웹팩 설정의 mode에 설정한 값이 여기에 들어간다. “development”를 설정했기 때문에 어플리케이션 코드에서 process.env.NODE\_ENV 변수로 접근하면 “development” 값을 얻을 수 있다.

app.js

console.log(process.env.NODE\_ENV) // "development"

이 외에도 웹팩 컴파일 시간에 결정되는 값을 전역 상수 문자열로 어플리케이션에 주입할 수 있다.

new webpack.DefinePlugin({

TWO: '1+1',

})

TWO라는 전역 변수에 1+1 이란 코드 조각을 넣었다. 실제 어플리케이션 코드에서 이것을 출력해보면 2가 나올 것이다.

app.js

console.log(TWO); // 2

코드가 아닌 값을 입력하려면 문자열화 한 뒤 넘긴다.

new webpack.DefinePlugin({

VERSION: JSON.stringify('v.1.2.3'),

PRODUCTION: JSON.stringify(false),

MAX\_COUNT: JSON.stringify(999),

'api.domain': JSON.stringify('<http://dev.api.domain.com>'),

})

app.js:

console.log(VERSION) // 'v.1.2.3'

console.log(PRODUCTION) // true

console.log(MAX\_COUNT) // 999

console.log(api.domain) // '<http://dev.api.domain.com>'

빌드 타임에 결정된 값을 어플리이션에 전달할 때는 이 플러그인을 사용하자.

6.3 HtmlWebpackPlugin

이번엔 써드 파티 패키지에 대해 알아보자. [HtmlWebpackPlugin](https://github.com/jantimon/html-webpack-plugin/)은 HTML 파일을 후처리하는데 사용한다. 빌드 타임의 값을 넣거나 코드를 압축할수 있다.

먼저 패키지를 다운로드 한다.

$ npm install -D html-webpack-plugin

이 플러그인으로 빌드하면 HTML파일로 아웃풋에 생성될 것이다. index.html 파일을 src/index.html로 옮긴뒤 다음과 같이 작성해 보자.

src/index.html:

<!DOCTYPE html>

<html>

<head>

<title>타이틀<%= env %></title>

</head>

<body>

<!-- 로딩 스크립트 제거 -->

<!-- <script src="dist/main.js"></script> -->

</body>

</html>

타이틀 부분에 ejs 문법을 이용하는데 <%= env %> 는 전달받은 env 변수 값을 출력한다. HtmlWebpackPlugin은 이 변수에 데이터를 주입시켜 동적으로 HTML 코드를 생성한다.

뿐만 아니라 웹팩으로 빌드한 결과물을 자동으로 로딩하는 코드를 주입해 준다. 때문에 스크립트 로딩 코드도 제거했다.

webpack.config.js:

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

module.exports {

plugins: [

new HtmlWebpackPlugin({

template: './src/index.html', // 템플릿 경로를 지정

templateParameters: { // 템플리셍 주입할 파라매터 변수 지정

env: process.env.NODE\_ENV === 'development' ? '(개발용)' : '',

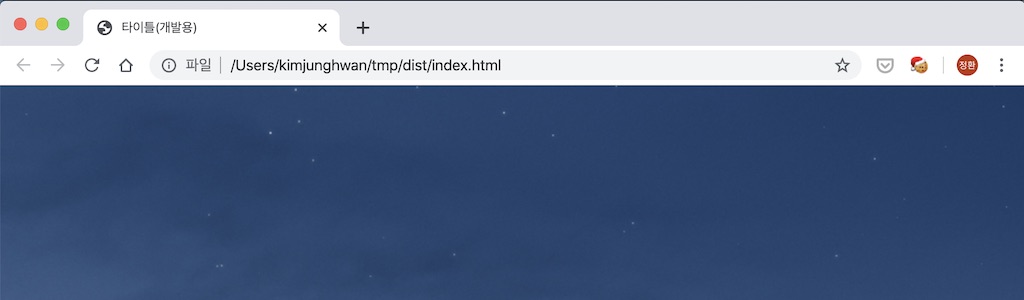
},

})

]

}

환경 변수에 따라 타이틀 명 뒤에 “(개발용)” 문자열을 붙이거나 떼거나 하도록 했다. NODE\_ENV=development 로 설정해서 빌드하면 빌드결과 “타이틀(개발용)”으로 나온다. NODE\_ENV=production 으로 설정해서 빌드하면 빌드결과 “타이틀”로 나온다.



개발 환경과 달리 운영 환경에서는 파일을 압축하고 불필요한 주석을 제거하는 것이 좋다.

webpack.config.js:

new HtmlWebpackPlugin({

minify: process.env.NODE\_ENV === 'production' ? {

collapseWhitespace: true, // 빈칸 제거

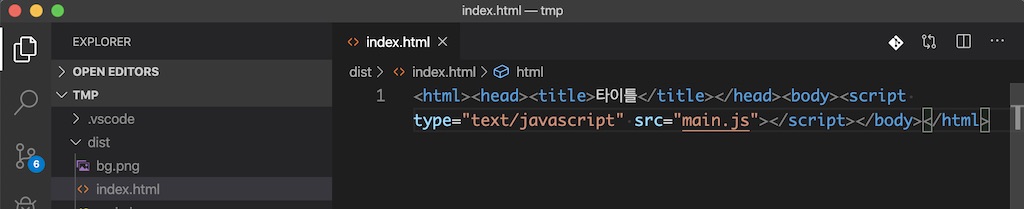
removeComments: true, // 주석 제거

} : false,

}

([문서에는minifiy 옵션이 웹팩 버전 3 기준으로 되어 있다](https://github.com/jantimon/html-webpack-plugin/issues/1094))

환경변수에 따라 minify 옵션을 켰다. NOE\_ENV=production npm run build로 빌드하면 아래처럼 코드가 압축된다. 물론 주석도 제거 되었다.



정적파일을 배포하면 즉각 브라우져에 반영되지 않는 경우가 있다. 브라우져 캐쉬가 원인일 경우가 있는데 이를 위한 예방 옵션도 있다.

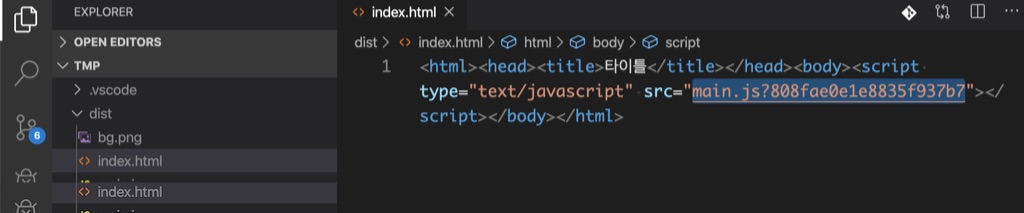
webpack.config.js:

new HtmlWebpackPlugin({

hash: true, // 정적 파일을 불러올때 쿼리문자열에 웹팩 해쉬값을 추가한다

})

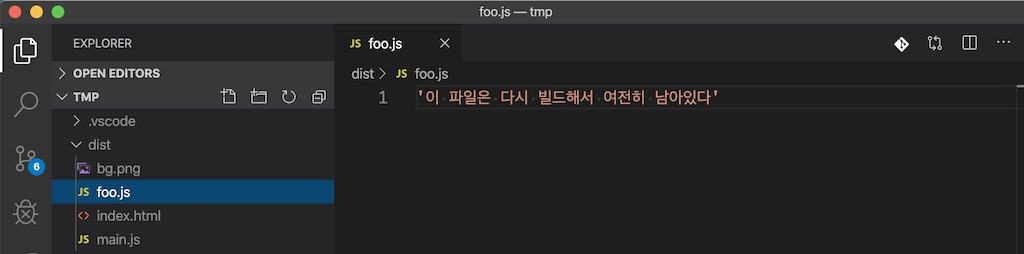
hash: true 옵션을 추가하면 빌드할 시 생성하는 해쉬값을 정적파일 로딩 주소의 쿼리 문자열로 붙여서 HTML을 생성한다.



6.4 CleanWebpackPlugin

[CleanWebpackPlugin](https://github.com/johnagan/clean-webpack-plugin)은 빌드 이전 결과물을 제거하는 플러그인이다. 빌드 결과물은 아웃풋 경로에 모이는데 과거 파일이 남아 있을수 있다. 이전 빌드내용이 덮여 씌여지면 상관없지만 그렇지 않으면 아웃풋 폴더에 여전히 남아 있을 수 있다.

임시로 아웃풋 폴더에 foo.js 파일을 만든 후 다시 빌드해 보자…… 파일이 남아 있다.



이러한 현상을 CleanWebpackPlugin으로 해결해 보자. 먼저 패키지를 설치한다.

$ npm install -D clean-webpack-plugin

웹팩 설정을 추가한다.

webpack.config.js:

const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {

plugins: [

new CleanWebpackPlugin(),

]

}

빌드 결과 foo.js가 깨끗히 사라졌다. 아웃풋 폴더인 dist 폴더가 모두 삭제된후 결과물이 생성되었기 때문이다.

6.5 MiniCssExtractPlugin

스타일시트가 점점 많아지면 하나의 자바스크립트 결과물로 만드는 것이 부담일 수 있다. 번들 결과에서 스트일시트 코드만 뽑아서 별도의 CSS 파일로 만들어 역할에 따라 파일을 분리하는 것이 좋다. 브라우져에서 큰 파일 하나를 내려받는 것 보다, 여러 개의 작은 파일을 동시에 다운로드하는 것이 더 빠르다.

개발 환경에서는 CSS를 하나의 모듈로 처리해도 상관없지만 프로덕션 환경에서는 분리하는 것이 효과적이다. [MiniCssExtractPlugin](https://github.com/webpack-contrib/mini-css-extract-plugin)은 CSS를 별로 파일로 뽑아내는 플러그인이다.

먼저 패키지를 설치한다.

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

웹팩 설정을 추가한다.

webpack.config.js:

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

module.exports = {

plugins: [

...(

process.env.NODE\_ENV === 'production'

? [ new MiniCssExtractPlugin({filename: `[name].css`}) ]

: []

),

],

}

프로덕션 환경일 경우만 이 플러그인을 추가했다. filename에 설정한 값으로 아웃풋 경로에 CSS 파일이 생성될 것이다.

개발 환경에서는 css-loader에의해 자바스크립트 모듈로 변경된 스타일시트를 적용하기위해 style-loader를 사용했다. 반면 프로덕션 환경에서는 별도의 CSS 파일으로 추출하는 플러그인을 적용했으므로 다른 로더가 필요하다.

module.exports = {

module: {

rules: [{

test: /\.css$/,

use: [

process.env.NODE\_ENV === 'production'

? MiniCssExtractPlugin.loader // 프로덕션 환경

: 'style-loader', // 개발 환경

'css-loader'

],

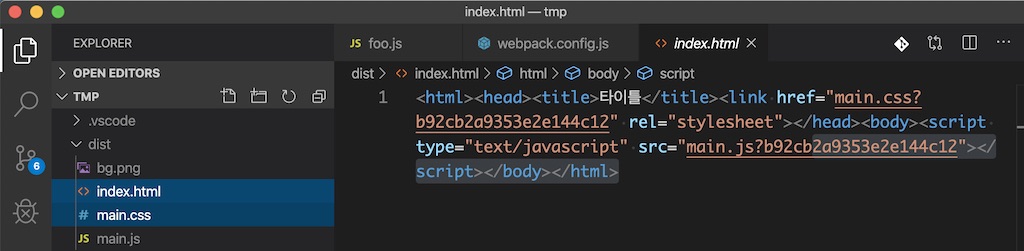
}]

}

}

플러그인에서 제공하는 MiniCssExtractPlugin.loader 로더를 추가한다.

NODE\_ENV=production npm run build로 결과를 확인해보자.



dist/main.css가 생성되었고 index.html에 이 파일을 로딩하는 코드가 추가되었다.

7. 정리

ECMAScript2015 이전에는 모듈을 만들기 위해 즉시실행함수와 네임스페이스 패턴을 사용했다. 이후 각 커뮤니티에서 모듈 시스템 스펙이 나왔고 웹팩은 ECMAScript2015 모듈시스템을 쉽게 사용하도록 돕는 역할을 한다.

엔트리포인트를 시작으로 연결되어 었는 모든 모듈을 하나로 합쳐서 결과물을 만드는 것이 웹팩의 역할이다. 자바스크립트 모듈 뿐만 아니라 스타일시트, 이미지 파일까지도 모듈로 제공해 주기 때문에 일관적으로 개발할 수 있다.

웹팩의 로더와 플러그인의 원리에 대해 살펴보았고 자주 사용하는 것들의 기본적인 사용법에 대해 익혔다.

[#webpack](http://jeonghwan-kim.github.io/tags/#webpack)

# 프론트엔드 개발환경의 이해: Babel

2019년 12월 22일

## 1. 배경

### 1.1 크로스 브라우징

사용하는 말이 달라서 바벨탑이 실패했듯이, 브라우져마다 사용하는 언어가 달라서 프론트엔트 코드는 일관적이지 못할 때가 많다. 스팩과 브라우져가 개선되고 있지만, 여전히 인터넷 익스플로러는 프라미스를 이해하지 못한다. 작년까지만 해도 사파리 최신 브라우져는 Promise.prototype.finally 메소드를 사용할 수 없었다. 프론트엔드 개발에서 크로스브라우징 이슈는 코드의 일관성을 해치고 초심자를 불안하게 만든다. 히브리어로 바벨이 ‘혼돈’이란 뜻인 것처럼 말이다.

크로스브라우징의 혼란을 해결해 줄 수 있는 것이 바벨이다. ECMAScript2015+로 작성한 코드를 모든 브라우져에서 동작하도록 호환성을 지켜준다. 타입스크립트, JSX처럼 다른 언어로 분류되는 것도 포함한다.

### 1.2 트랜스파일과 빌드

이렇게 변환하는 것을 “트랜스파일” 한다라고 표현한다. 변환 전후의 추상화 수준이 다른 빌드와는 달리 트랜스파일은 추상화 수준을 유지한 상태로 코드를 변환한다. 타입스크립트 → 자바스크립트, JSX → 자바스크립트처럼 트랜스파일 후에도 여전히 코드를 읽을 수 있다.

요즘에는 이 둘을 구분하지 않고 사용하는 것 같다.

## 2. 바벨의 기본 동작

바벨은 ECMAScript2015 이상의 코드를 적당한 하위 버전으로 바꾸는 것이 주된 역할이다. 이렇게 바뀐 코드는 인터넷 익스프로러나 구버전 브라우져처럼 최신 자바스크립트 코드를 이해하지 못하는 환경에서도 잘 동작한다.

바벨을 이용해 아래 코드를 인터넷 익스플로러가 이해할 수 있는 코드로 바꿔 보겠다.

*// src/app.js:*

**const** alert **=** msg **=>** window.alert(msg);

먼저 바벨 최신 버전를 설치한다. 터미널 도구를 사용하기 위해 커맨드라인 도구도 함께 설치하자.

npm install -D @babel/core @babel/cli

설치를 완료후 node\_modules/.bin 폴더에 추가된 바벨 명령어를 사용할 수 있다.

npx babel app.js

const alert = msg => window.alert(msg);

바벨은 세 단계로 빌드를 진행한다.

1. 파싱(Parsing)
2. 변환(Transforming)
3. 출력 (Printing)

코드를 읽고 추상 구문 트리(AST)로 변환하는 단계를 **“파싱”**이라고 한다. 이것은 빌드 작업을 처리하기에 적합한 자료구조인데 컴파일러 이론에 사용되는 개념이다. 추상 구문 트리를 변경하는 것이 **“변환”** 단계이다. 실제로 코드를 변경하는 작업을 한다. 변경된 결과물을 **“출력”**하는 것을 마지막으로 바벨은 작업을 완료한다.

그런데…… 결과를 보면 빌드 이전과 변한게 하나도 없다.

## 3. 플러그인

기본적으로 바벨은 코드를 받아서 코드를 반환한다. 바벨 함수를 정의한다면 이런 모습이 될 것이다.

**const** babel **=** code **=>** code;

바벨은 파싱과 출력만 담당하고 변환 작업은 다른 녀석이 처리하데 이것을 **“플러그인”** 이라고 부른다.

### 3.1 커스텀 플러그인

플러그인을 직접 만들면서 동작이 원리를 살펴 보겠다. myplugin.js 라는 파일을 아래처럼 만들어 보자(출처: [바벨 홈페이지의 예제 코드](https://babeljs.io/docs/en/plugins#plugin-development)).

*// myplugin.js:*

module.exports **=** **function** myplugin() {

**return** {

visitor: {

Identifier(path) {

**const** name **=** path.node.name;

*// 바벨이 만든 AST 노드를 출력한다*

console.log('Identifier() name:', name)

*// 변환작업: 코드 문자열을 역순으로 변환한다*

path.node.name **=** name

.split("")

.reverse()

.join("");

}

},

};

}

플러그인 형식은 visitor 객체를 가진 함수를 반환해야 한다. 이 객체는 바벨이 파싱하여 만든 추상 구문 트리(AST)에 접근할 수 있는 메소드를 제공한다. 그중 Identifier() 메소드의 동작 원리를 살펴보는 코드다.

플러그인 사용법을 알아보자.

npx babel --help

--plugins [list] A comma-separated list of plugin names.

--plugins 옵션에 플러그인을 추가하면 된다.

npx babel app.js --plugins ./myplugin.js

Identifier() name: alert

Identifier() name: msg

Identifier() name: window

Identifier() name: alert

Identifier() name: msg

const trela = gsm => wodniw.trela(gsm);

Identifier() 메소드로 들어온 인자 path에 접근하면 코드 조각에 접근할 수 있는 것 같다. path.node.name의 값을 변경하는데 문자를 뒤집는 코드다. 결과의 마지막 줄에서 보는것 처럼 이 코드의 문자열 순서가 역전되었다.

우리가 하려는것은 ECMASCript2015로 작성한 코드를 인터넷 익스플로러에서 돌리는 것이다. 먼저 const 코드를 var로 변경하는 플러그인을 만들어 보겠다.

*// myplugin.js:*

module.exports **=** **function** myplugin() {

**return** {

visitor: {

*//* [*https://github.com/babel/babel/blob/master/packages/babel-plugin-transform-block-scoping/src/index.js#L26*](https://github.com/babel/babel/blob/master/packages/babel-plugin-transform-block-scoping/src/index.js#L26)

VariableDeclaration(path) {

console.log('VariableDeclaration() kind:', path.node.kind); *// const*

**if** (path.node.kind **===** 'const') {

path.node.kind **=** 'var'

}

},

},

};

}

이번에는 vistor 객체에 VariableDeclaration() 메소드를 정의했다. path에 접근해 보면 키워드가 잡히는 걸 알 수 있다. path.node.kind가 const 일 경우 var로 변환하는 코드다.

이 플러그인으로 다시 빌드해보자.

npx babel app.js --plugins ./myplugin.js

VariableDeclaration() kind: const

var alert = msg => window.alert(msg);

마지막 줄에 보면 const 코드가 var로 변경되었다.

### 3.2 플러그인 사용하기

이러한 결과를 만드는 것이 [block-scoping](https://babeljs.io/docs/en/babel-plugin-transform-block-scoping) 플러그인이다. const, let 처럼 블록 스코핑을 따르는 예약어를 함수 스코핑을 사용하는 var 변경한다.

NPM 패키지로 제공하는 플러그인을 설치하고,

npm install -D @babel/plugin-transform-block-scoping

설치한 플러그인을 사용해보면,

npx babel app.js --plugins @babel/plugin-transform-block-scoping

var alert = msg => window.alert(msg);

커스텀 플러그인과 같은 결과를 보인다.

인터넷 익스플로러는 화살표 함수도 지원하지 않는데 [arrow-functions](https://babeljs.io/docs/en/babel-plugin-transform-arrow-functions) 플러그인을 이용해서 일반 함수로 변경할 수 있다.

npm install -D @babel/plugin-transform-arrow-functions

npx babel app.js \

--plugins @babel/plugin-transform-block-scoping \

--plugins @babel/plugin-transform-arrow-functions

var alert = function (msg) {

return window.alert(msg);

};

ECMAScript5에서부터 지원하는 엄격 모드를 사용하는 것이 안전하기 때문에 "use strict" 구문을 추가해야 겠다. [strict-mode](https://babeljs.io/docs/en/babel-plugin-transform-strict-mode) 플러그인을 사용하자.

그전에 커맨드라인 명령어가 점점 길어지기 때문에 설정 파일로 분리하는 것이 낫겠다. 웹팩 webpack.config.js를 기본 설정파일로 사용하듯 바벨도 [babel.config.js](https://babeljs.io/docs/en/config-files#project-wide-configuration)를 사용한다.

프로젝트 루트에 babel.config.js 파일을 아래와 같이 작성하자.

*// babel.config.js:*

module.exports **=** {

plugins: [

"@babel/plugin-transform-block-scoping",

"@babel/plugin-transform-arrow-functions",

"@babel/plugin-transform-strict-mode",

]

}

커맨드라인에서 사용한 block-scoping, arrow-functions 플러그인을 설정 파일로 옮겼는데 plugins 배열에 추가하는 방식이다. strict-mode 플러그인을 마지막 줄에 추가했다.

다시 빌드해보자.

npx babel app.js

"use strict";

var alert = function (msg) {

return window.alert(msg);

};

상단에 “use strict” 구문이 추가되어 엄격모드가 활성화 되었다. 이제야 비로소 인터넷 익스플로러에서 안전하게 동작하는 코드로 트랜스파일하였다.

이처럼 변환을 위한 플러그인 목록은 공식 문서의 [Plugins](https://babeljs.io/docs/en/plugins) 페이지에서 확인할 수 있다.

## 4. 프리셋

ECMAScript2015+으로 코딩할 때 필요한 플러그인을 일일이 설정하는 일은 무척 지난한 일이다. 코드 한 줄 작성하는데도 세 개 플러그인 세팅을 했으니 말이다. 목적에 맞게 여러가지 플러그인을 세트로 모아놓은 것을 **“프리셋”**이라고 한다.

### 4.1 커스텀 프리셋

사용한 세 개 플러그인을 하나의 프리셋으로 만들어 보겠다. mypreset.js 파일을 다음과 같이 작성하자.

*// mypreset.js*

module.exports **=** **function** mypreset() {

**return** {

plugins: [

"@babel/plugin-transform-arrow-functions",

"@babel/plugin-transform-block-scoping",

"@babel/plugin-transform-strict-mode",

],

}

}

plugins 배열에 사용한 세 개 플러그인을 담았다.

프리셋을 사용하기 위해 바벨 설정을 약간 수정한다.

*// babel.config.js*

module.exports **=** {

presets: [

'./mypreset.js'

],

}

플러그인 세팅 코드를 제거하고 presets에 방금 만든 mypreset.js를 추가했다. 실행해보면 동일한 결과를 출력할 것이다.

### 4.2 프리셋 사용하기

이처럼 바벨은 목적에 따라 몇 가지 [프리셋](https://babeljs.io/docs/en/presets)을 제공한다.

* preset-env
* preset-flow
* preset-react
* preset-typescript

preset-env는 ECMAScript2015+를 변환할 때 사용한다. 바벨 7 이전 버전에는 연도별로 각 프리셋을 제공했지만(babel-reset-es2015, babel-reset-es2016, babel-reset-es2017, babel-reset-latest) 지금은 env 하나로 합쳐졌다. 무척 맘에드는 부분이다.

preset-flow, preset-react, preset-typescript는 flow, 리액트, 타입스크립트를 변환하기 위한 프리셋이다.

인터넷 익스플로러 지원을 위해 env 프리셋을 사용해 보자. 먼저 패키지를 다운로드한다.

npm install -D @babel/preset-env

설치한 바벨 설정을 조금만 더 바꿔본다.

*// babel.config.js:*

module.exports **=** {

presets: [

'@babel/preset-env'

]

}

그리고 빌드하면,

npx babel app.js

"use strict";

var alert = function alert(msg) {

return window.alert(msg);

};

우리가 만든 mypreset.js와 같은 결과를 출력한다.

## 5. env 프리셋 설정과 폴리필

과거에 제공했던 연도별 프리셋을 사용해 본 경험이 있다면 까다롭고 헷갈리는 설정 때문에 애를 먹었을지도 모르겠다. 그에 비해 env 프리셋은 무척 단순하고 직관적인 사용법을 제공한다.

### 5.1 타겟 브라우져

우리 코드가 크롬 최신 버전(2019년 12월 기준)만 지원하다고 하자. 그렇다면 인터넷 익스플로러를 위한 코드 변환은 불필요하다. target 옵션에 브라우져 버전명만 지정하면 env 프리셋은 이에 맞는 플러그인들을 찾아 최적의 코드를 출력해 낸다.

*// babel.config.js*

module.exports **=** {

presets: [

[

'@babel/preset-env',

{

targets: {

chrome: '79', *// 크롬 79까지 지원하는 코드를 만든다*

}

}

]

]

}

npx babel app.js

"use strict";

const alert = msg => window.alert(msg);

크롬은 블록 스코핑과 화살표 함수를 지원하기 때문에 코드를 변환하지 않고 이러한 결과물을 만들었다. 만약 인터넷 익스플로러도 지원해야 한다면 바벨 설정에 브라우져 정보만 하나 더 추가하면 된다.

*// babel.config.js :*

module.exports **=** {

presets: [

[

'@babel/preset-env',

{

targets: {

chrome: '79',

ie: '11' *// ie 11까지 지원하는 코드를 만든다*

}

}

]

]

}

### 5.2 폴리필

이번엔 변환과 조금 다른 플리필에 대해 알아보자.

ECMASCript2015의 Promise 객체를 사용하는 코드다.

*// app.js:*

**new** Promise();

바벨로 처리하면 어떤 결과가 나올까?

npx babel app.js

"use strict";

new Promise();

env 프리셋으로 변환을 시도했지만 Promise 그대로 변함이 없다. target에 ie 11을 설정하고 빌드한 것인데 인터넷 익스플로러는 여전히 프라미스를 해석하지 못하고 에러를 던진다.

promise-error-in-ie

브라우져는 현재 스코프부터 시작해 전역까지 Promise라는 이름을 찾으려고 시도할 것이다. 그러나 스코프 어디에도 Promise란 이름이 없기 때문에 레퍼런스 에러를 발생하고 프로그램이 죽은 것이다.

플러그인이 프라미스를 ECMAScript5 버전으로 변환할 것으로 기대했는데 예상과 다르다. 바벨은 ECMAScript2015+를 ECMAScript5 버전으로 변환할 수 있는 것만 빌드한다. 그렇지 못한 것들은 “**폴리필**“이라고 부르는 코드조각을 추가해서 해결한다.

가령 ECMAScript2015의 블록 스코핑은 ECMASCript5의 함수 스코핑으로 대체할 수 있다. 화살표 함수도 일반 함수로 대체할 수 있다. 이런 것들은 바벨이 변환해서 ECMAScript5 버전으로 결과물을 만든다.

한편 프라미스는 ECMAScript5 버전으로 대체할 수 없다. 다만 ECMAScript5 버전으로 구현할 수는 있다(참고: [core-js promise](https://github.com/zloirock/core-js/blob/master/packages/core-js/modules/es.promise.js)).

env 프리셋은 폴리필을 지정할 수 있는 옵션을 제공한다.

*// babel.config.js:*

module.exports **=** {

presets: [

[

'@babel/preset-env',

{

useBuiltIns: 'usage', *// 폴리필 사용 방식 지정*

corejs: { *// 폴리필 버전 지정*

version: 2

}

},

],

],

};

useBuiltIns는 어떤 방식으로 폴리필을 사용할지 설정하는 옵션이다. “usage” , “entry”, false 세 가지 값을 사용하는데 기본값이 false 이므로 폴리필이 동작하지 않았던 것이다. 반면 usage나 entry를 설정하면 폴리필 패키지 중 core-js를 모듈로 가져온다(이전에 사용하던 babel/polyfile은 바벨 7.4.0부터 사용하지 않음).

corejs 모듈의 버전도 명시하는데 기본값은 2다. 버전 3과 차이는 확실히 잘 모르겠다. 이럴 땐 그냥 기본값을 사용하는 편이다.

자세한 폴리필 옵션은 바벨 문서의 [useBuiltIns](https://babeljs.io/docs/en/babel-preset-env#usebuiltins)와 [corejs](https://babeljs.io/docs/en/babel-preset-env#corejs) 섹션을 참고하자.

폴리필이 추가된 결과물을 확인해 보자.

npx babel src/app.js

"use strict";

require("core-js/modules/es6.promise");

require("core-js/modules/es6.object.to-string");

new Promise();

core-js 패키지로부터 프라미스 모듈을 가져오는 임포트 구문이 상단에 추가되었다. 이제야 비로소 인터넷 익스플로러에서 안전하게 돌아가는 결과물을 만들었다.

## 6. 웹팩으로 통합

실무 환경에서는 바벨을 직접 사용하는 것보다는 웹팩으로 통합해서 사용하는 것이 일반적이다. 로더 형태로 제공하는데 [babel-loader](https://github.com/babel/babel-loader)가 그것이다.

먼저 패키지를 설치하고,

npm install -D babel-loader

웹팩 설정에 로더를 추가한다.

*// webpack.config.js:*

module.exports **=** {

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

loader: 'babel-loader', *// 바벨 로더를 추가한다*

},

]

},

}

.js 확장자로 끝나는 파일은 babel-loader가 처리하도록 설정했다. 사용하는 써드파티 라이브라리가 많을수록 바벨 로더가 느리게 동작할 수 있는데 node\_modules 폴더를 로더가 처리하지 않도록 예외 처리했다([참고](https://github.com/babel/babel-loader#babel-loader-is-slow)).

폴리필 사용 설정을 했다면 core-js도 설치해야한다. 웹팩은 바벨 로더가 만든 아래 코드를 만나면 core-js를 찾을 것이기 때문이다.

require("core-js/modules/es6.promise");

require("core-js/modules/es6.object.to-string");

버전 2로 패키지를 추가하자.

npm i core-js@2

그리고 웹팩으로 빌드하면,

npm run build

> webpack

Hash: a30cff5fbf53027423a0

Version: webpack 4.41.2

Time: 718ms

Built at: 2019. 12. 16. 오전 8:52:05

Asset Size Chunks Chunk Names

main.js 59.7 KiB main [emitted] main

Entrypoint main = main.js

[./src/app.js] 166 bytes {main} [built]

미리 등록해 놓은 NPM build 스크립트의 webpack 명령어가 동작한다. ./src/app.js의 엔트리 포인트가 바벨 로더에 의해 빌드되고 결과물이 dist/main.js로 옮겨졌다.

cat ./dist/main.js | grep 'var alert' -A 5

var alert = function alert(msg) {

return window.alert(msg);

};

new Promise();

웹팩으로 번들링되면서 변경된 부분 찾기가 어려울수 있는데 grep으로 변경되 부분만 확인했다.

## 7. 정리

바벨은 일관적인 방식으로 코딩하면서, 다양한 브라우져에서 돌아가는 어플리케이션을 만들기 위한 도구다.

바벨의 코어는 파싱과 출력만 담당하고 변환 작업은 플러그인이 처리한다.

여러 개의 플러그인들을 모아놓은 세트를 프리셋이라고 하는데 ECMAScript+ 환경은 env 프리셋을 사용한다.

바벨이 변환하지 못하는 코드는 폴리필이라 부르는 코드조각을 불러와 결과물에 로딩해서 해결한다.

babel-loader로 웹팩과 함께 사용하면 훨씬 단순하고 자동화된 프론트엔드 개발환경을 갖출 수 있다.

# 프론트엔드 개발환경의 이해: 린트

2019년 12월 30일

## 1. 배경

오래된 스웨터의 보푸라기 같은 것을 린트(Lint)라고 부른다. 보푸라기가 많으면 옷이 보기 좋지 않은데 코드에서도 이런 보프라기가 있다. 들여쓰기를 맞추지 않은 경우, 선언한 변수를 사용하지 않은 경우……

보프라기 있는 옷을 입을 수는 있듯이 이러한 코드로 만든 어플리케이션도 동작은 한다. 그러나 코드의 가독성이 떨어지고 점점 유지보수하기 어려운 애물단지가 되어버리기 일쑤다.

보푸라기를 제거하는 린트 롤러(Lint roller)처럼 코드의 오류나 버그, 스타일 따위를 점검하는 것을 [린트(Lint) 혹은 린터(Linter)](https://en.wikipedia.org/wiki/Lint_(software))라고 부른다.

### 1.1 린트가 필요한 상황

아래 코드 유심히 보자. console.log() 함수를 실행하고 다음 줄에서 즉시 실행함수를 실행하려는 코드다.

console.log()

(**function**() {})()

하지만 이 코드를 브라우져에서 실행해 보면 TypeError가 발생한다. 브라우져는 코드에 세미콜론를 자동으로 넣는 과정(ASI)을 수행하는데, 위와 같은 경우는 우리의 의도대로 해석하지 못하고 아래 코드로 해석한다([Rules of Automatic Semicolon Insertion](http://www.ecma-international.org/ecma-262/7.0/index.html#sec-rules-of-automatic-semicolon-insertion)을 참고).

console.log()(**function**(){})();

console.log()가 반환하는 값이 함수가 아닌데 함수 호출을 시도했기 때문에 타입에러가 발생할 것이다. 모든 문장에 세미콜론을 붙였다면, 혹은 즉시 함수호출 앞에 세미콜론을 붙였다면 예방할 수 있는 버그다.

린트는 코드의 가독성을 높이는 것 뿐만 아니라 동적 언어의 특성인 런타임 버그를 예방하는 역할도 한다.

## 2. ESLint

### 2.1 기본 개념

ESLint는 ECMAScript 코드에서 문제점을 검사하고 일부는 더 나은 코드로 정정하는 린트 도구 중의 하나다. 코드의 가독성을 높이고 잠재적인 오류와 버그를 제거해 단단한 코드를 만드는 것이 목적이다. 과거 JSLint, JSHint에 이어서 최근에는 ESList를 많이 사용하는 편이다.

코드에서 검사하는 항목을 크게 분류하면 아래 두 가지다.

* 포맷팅
* 코드 품질

**포맷팅**은 일관된 코드 스타일을 유지하도록 하고 개발자로 하여금 쉽게 읽히는 코드를 만들어 준다. 이를 테면 “들여쓰기 규칙”, “코드 라인의 최대 너비 규칙”을 따르는 코드가 가독성이 더 좋다.

한편, **코드 품질**은 어플리케이션의 잠재적인 오류나 버그를 예방하기 위함이다. 사용하지 않는 변수 쓰지 않기, 글로벌 스코프 함부로 다루지 않기 등이 오류 발생 확률을 줄여 준다.

린트 도구를 사용해서 코드를 검사하고 더 나아가 단단한 하고 읽기 좋은 코드를 만드는 방법을 알아보자.

### 2.2 설치 및 사용법

먼저 노드 패키지로 제공되는 ESLint 도구를 다운로드 한다.

npm i -D eslint

환경설정 파일을 프로젝트 최상단에 생성한다.

*// .eslintrc.js*

module.exports **=** {}

빈 객체로 아무런 설정 없이 모듈만 만들었다. ESLint로 코드를 검사 하면

npx eslint app.js

아무런 결과를 출력하지 않고 프로그램을 종료한다.

### 2.3 규칙(Rules)

ESLint는 검사 규칙을 미리 정해 놓았다. 문서의 [Rules](https://eslint.org/docs/rules/) 메뉴에서 규칙 목록을 확인할 수 있다.

우리가 우려했던 문제와 관련된 규칙은 [no-unexpected-multiline](https://eslint.org/docs/rules/no-unexpected-multiline)이다. 설정 파일의 rules 객체에 이 규칙을 추가한다.

*// .eslintrc.js*

module.exports **=** {

rules: {

"no-unexpected-multiline": "error"

}

}

규칙에 설정하는 값은 세 가지다. “off”나 0은 끔, “warn”이나 1은 경고, “error”나 2는 오류. 설정한 규칙에 어긋나는 코드를 발견하면 오류를 출력하도록 했다.

다시 검사해 보자.

npx eslint app.js

2:1 error Unexpected newline between function and ( of function call no-unexpected-multiline

✖ 1 problem (1 error, 0 warnings)

예상대로 에러가 발생하고 코드 위치와 위반한 규칙명을 알려준다. 함수와 함수 호출의 괄호 “(“ 사이에 줄바꿈이 있는데 이것이 문제라고 한다. 코드 앞에 세미콜론을 넣거나 모든 문의 끝에 세미콜론을 넣어 문제를 해결할 수 있다. 수정한 다음 다시 검사하면 검사에 통과할 것이다.

### 2.4 자동으로 수정할 수 있는 규칙

자바스크립트 문장 뒤에 세미콜론을 여러 개 중복 입력해도 어플리케이션은 동작한다. 그러나 이것은 코드를 읽기 어렵게 하는 장애물일 뿐이다. 이렇게 작성한 코드가 있다면 실수로 입력한게 틀림 없다.

이 문제와 관련된 규칙은 [no-extra-semi](https://eslint.org/docs/rules/no-extra-semi) 규칙이다.

문장 뒤에 세미콜론을 더 추가한 뒤,

*// app.js*

console.log();; *// 세미콜론 연속 두 개 붙임*

린트 설정에 no-extra-semi 규칙을 추가하고,

*// .eslintrc.js*

module.exports **=** {

rules: {

"no-extra-semi": "error",

}

}

코드를 검사하면 오류를 출력한다.

npx eslint app.js

1:15 error Unnecessary semicolon no-extra-semi

✖ 1 problem (1 error, 0 warnings)

1 error and 0 warnings potentially fixable with the `--fix` option.

마지막 줄의 메세지를 보면 이 에러는 “잠재적으로 수정가능(potentially fixable)”하다고 말한다. --fix 옵션을 붙여 검사해보면,

npx eslint app.js --fix

검사 후 오류가 발생하면 코드를 자동으로 수정한다.

이렇듯 ESLint 규칙에는 수정 가능한 것과 그렇지 못한 것이 있다.

[규칙 목록](https://eslint.org/docs/rules/) 중 왼쪽에 렌치 표시가 붙은 것이 --fix 옵션으로 자동 수정할 수 있는 규칙이다.

### 2.5 Extensible Config

이러한 규칙을 여러게 미리 정해 놓은 것이 eslint:recommended 설정이다. [규칙 목록](https://eslint.org/docs/rules/) 중에 왼쪽에 체크 표시되어 있는것이 이 설정에서 활성화되어 있는 규칙이다.

이것을 사용하려면 extends 설정을 추가한다.

*// .eslintrc.js*

module.exports **=** {

extends: [

"eslint:recommended", *// 미리 설정된 규칙 세트을 사용한다*

],

}

만약 이 설정 외에 규칙이 더 필요하다면 rules 속성에 추가해서 확장할 수 있다.

ESLint에서 기본으로 제공하는 설정 외에 자주 사용하는 두 가지가 있다.

* airbnb
* standard

**airbnb** 설정은 [airbnb 스타일 가이드](https://github.com/airbnb/javascript)를 따르는 규칙 모음이다. [eslint-config-airbnb-base](https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb-base) 패키지로 제공된다.

**standard** 설정은 [자바스크립트 스탠다드 스타일](https://standardjs.com/)을 사용한다. [eslint-config-standard](https://github.com/standard/eslint-config-standard) 패키지로 제공된다.

### 2.6 초기화

사실 이러한 설정은 --init 옵션을 추가하면 손쉽게 구성할 수 있다.

npx eslint --init

? How would you like to use ESLint?

? What type of modules does your project use?

? Which framework does your project use?

? Where does your code run?

? How would you like to define a style for your project?

? Which style guide do you want to follow?

? What format do you want your config file to be in?

대화식 명령어로 진행되는데 모듈 시스템을 사용하는지, 어떤 프레임웍을 사용하는지, 어플리케이션이 어떤 환경에서 동작하는지 등에 답하면 된다. 답변에 따라 .eslintrc 파일을 자동으로 만들 수 있다.

## 3. Prettier

프리티어는 코드를 **“더”** 예쁘게 만든다. ESLint의 역할 중 포매팅과 겹치는 부분이 있지만 프리티터는 좀 더 일관적인 스타일로 코드를 다듬는다. 반면 코드 품질과 관련된 기능은 하지 않는 것이 ESLint와 다른 점이다.

### 3.1 설치 및 사용법

프리티어 패키지를 다운로드 하고

npm i -D prettier

코드를 아래 처럼 작성한다.

*// app.js*

console.log('hello world')

Prettier로 검사해 보자.

npx prettier app.js --write

--write 옵션을 추가하면 파일을 재작성한다. 그렇지 않을 경우 결과를 터미널에 출력한다.

변경된 모습을 보면,

*// app.js*

console.log("Hello world");

작은 따옴표를 큰 따옴표로 변경했다. 문장 뒤에 세미콜론도 추가했다. 프리티어는 ESLint와 달리 규칙이 미리 세팅되어 있기 때문에 설정 없이도 바로 사용할 수 있다.

### 3.2 포매팅(더 예쁘게)

다음 코드를 보자.

*// app.js*

console.log('----------------매 우 긴 문 장 입 니 다 80자가 넘 는 코 드 입 니 다.----------------')

ESLint는 [max-len](https://eslint.org/docs/rules/max-len) 규칙을 이용해 위 코드를 검사하고 결과만 알려 줄 뿐 수정하는 것은 개발자의 몫이다. 반면 프리티어는 어떻게 수정해야할지 알고 있기 때문에 아래처럼 코드를 다시 작성한다.

*// app.js*

console.log(

"----------------매 우 긴 문 장 입 니 다 80자가 넘 는 코 드 입 니 다.----------------"

);

아래 코드는 어떻게 변환할까?

foo(reallyLongArg(), omgSoManyParameters(), IShouldRefactorThis(), isThereSeriouslyAnotherOne());

프리티어는 코드를 문맥을 어느 정도 파악하고 상황에 따라 최적의 모습으로 스타일을 수정한다.

foo(

reallyLongArg(),

omgSoManyParameters(),

IShouldRefactorThis(),

isThereSeriouslyAnotherOne()

);

더 멋진 예제도 있는데 프리티어를 만든 [James Long의 글](https://jlongster.com/A-Prettier-Formatter)을 참고하자. 이러한 포매팅 품질은 ESLint보다는 프리티어가 훨씬 좋은 결과를 만든다. 사람에게 더 친숙하도록 말이다.

### 3.3 통합방법

여전히 ESLint를 사용해야 하는 이유는 남아 있다. 포맷팅은 프리티어에게 맡기더라도 코드 품질과 관련된 검사는 ESLint의 몫이기 때문이다. 따라서 이 둘을 같이 사용하는 것이 최선이다. 프리티어는 이러한 ESLint와 통합 방법을 제공한다. [eslint-config-prettier](https://github.com/prettier/eslint-config-prettier) 는 프리티어와 충돌하는 ESLint 규칙을 끄는 역할을 한다. 둘 다 사용하는 경우 규칙이 충돌하기 때문이다.

패키지를 설치한뒤,

npm i -D eslint-config-prettier

설정파일의 extends 배열에 추가한다.

*// .eslintrc.js*

{

extends: [

"eslint:recommended",

"eslint-config-prettier"

]

}

예를 들어 ESLint는 중복 세미콜론 사용을 검사한다. 이것을 프리티어도 마찬가지다. 따라서 어느 한쪽에서는 규칙을 꺼야하는데 eslint-config-prettier를 extends 하면 중복되는 ESLint 규칙을 비활성화 한다.

**var** foo **=** '' *// 사용하지 않은 변수. ESLint가 검사*

console.log();;;;;;; *// 중복 세미콜론 사용. 프리티어가 자동 수정*

ESLint는 중복된 포매팅 규칙을 프리티어에게 맞기고 나머지 코드 품질에 관한 검사만 한다. 따라서 아래처럼 두 개를 동시에 실행해서 코드를 검사한다.

npx prettier app.js --write && npx eslint app.js --fix

1:5 error 'foo' is assigned a value but never used no-unused-vars

✖ 1 problem (1 error, 0 warnings)

프리터에서 의해 코드가 아래과 같이 포매팅 되었고

**var** foo **=** "";

console.log();

ESlint에 의해 코드 품질과 관련된 오류(no-unused-vars)를 리포팅한다.

한편, [eslint-plugin-prettier](https://github.com/prettier/eslint-plugin-prettier)는 프리티어 규칙을 ESLint 규칙으로 추가하는 플러그인이다. 프리티어의 모든 규칙이 ESLint로 들어오기 때문에 ESLint만 실행하면 된다.

패키지를 설치하고

npm i -D eslint-plugin-prettier

설정 파일에서 pulugins와 rules에 설정을 추가한다.

*// .eslintrc.js*

{

plugins: [

"prettier"

],

rules: {

"prettier/prettier": "error"

},

}

프리티어의 모든 규칙을 ESLint 규칙으로 가져온 설정이다. 이제는 ESLint만 실행해도 프리티어 포매팅 기능을 가져갈 수 있다.

npx eslint app.js --fix

프리티어는 이 두 패키지를 함께 사용하는 [단순한 설정](https://prettier.io/docs/en/integrating-with-linters.html)을 제공하는데 아래 설정을 추가하면 된다.

*// .eslintrc.js*

{

"extends": [

"eslint:recommended",

"plugin:prettier/recommended"

]

}

## 4. 자동화

린트는 코딩할 때마다 수시로 실행해야하는데 이러한 일은 자동화 하는 것이 좋다. “깃 훅을 사용하는 방법”과 “에디터 확장 도구”를 사용하는 방법을 각각 소개한다.

### 4.1 변경한 내용만 검사

소스 트래킹 도구로 깃을 사용한다면 깃 훅을 이용하는 것이 좋다. 커밋 전, 푸시 전 등 깃 커맨드 실행 시점에 끼여들수 있는 훅을 제공한다. [husky](https://github.com/typicode/husky)는 깃 훅을 쉽게 사용할 수 있는 도구다. (Git 2.13.0 이상 버전을 지원) 커밋 메세지 작성전에 끼어들어 린트로 코드 검사 작업을 추가하면 좋겠다.

먼저 패키지를 다운로드 한다.

npm i -D husky

허스키는 패키지 파일에 설정을 추가한다.

package.json:

{

"husky": {

"hooks": {

"pre-commit": "echo \"이것은 커밋전에 출력됨\""

}

}

}

훅이 제대로 동작하는지 빈 커밋을 만들어 보자.

git commit --allow-empty -m "빈 커밋"

husky > pre-commit (node v13.1.0)

이것은 커밋전에 출력됨 ----> 깃 훅이 동작함

[master db8b4b8] empty

pre-commit에 설정한 내용이 출력되었다. 출력 대신에 린트 명령어로 대체하면 커밋 메세지 작성 전에 린트를 수행할 수 있겠다.

package.json:

{

"husky": {

"hooks": {

"pre-commit": "eslint app.js --fix"

}

}

}

만약 린트 수행중 오류를 발견하면 커밋 과정은 취소된다. 린트를 통과하게끔 코드를 수정해야만 커밋할 수 있는 환경이 되었다.

한편, 코드가 점점 많아지면 커밋 작성이 느려질 수 있는데 커밋전에 모든 코드를 린트로 검사하는 시간이 소요되기 때문이다. 커밋시 변경된 파일만 린트로 검사하면 더 좋지 않을까? [lint-staged](https://github.com/okonet/lint-staged)는 변경된(스테이징된) 파일만 린트를 수행하는 도구다.

패키지를 설치하고,

npm i -D lint-staged

패키지 파일에 설정을 추가한다.

package.json:

{

"lint-staged": {

"\*.js": "eslint --fix"

}

}

내용이 변경된 파일 중에 .js 확장자로 끝나는 파일만 린트로 코드 검사를 한다.

pre-commit 훅도 아래처럼 변경한다.

package.json:

{

"husky": {

"hooks": {

"pre-commit": "lint-staged"

}

},

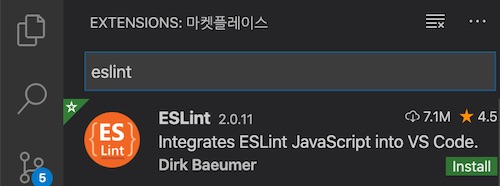
}

커밋 메세지 작성전에 lint-staged를 실행할 것이다. 이제 커밋하면 모든 파일을 검사하는 것이 아니라 변경되거나 추가된 파일만 검사한다. 커밋 과정이 훨씬 가벼워질 것이다.

### 4.2 에디터 확장도구

코딩할때 실시간으로 검사하는 방법도 있다. vs-code의 eslint와 prettier 익스텐션이 그러한 기능을 제공한다. 우리는 프리티어 규칙을 ESLint와 통합했기 때문에 ESLint 익스텐션을 사용해 보겠다.

먼저 [ESLint 익스텐션](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint) 부터 설치해 보자.



설치를 마친 뒤 eslint를 활성화 설정을 추가한다.

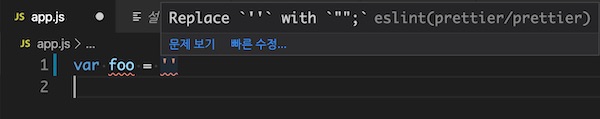
.vscode/settings.json:

{

"eslint.enable": **true**,

}

설치하면 자동으로 ESLint 설정파일을 읽고 파일을 검사한다.



툴팁 메뉴를 클릭해서 문제를 수정한다.

에디터 설정중 저장시 액션을 추가할 수 있는데 ESLint로 코드를 정정할 수 있다.

.vscode/settings.json:

{

"editor.codeActionsOnSave": {

"source.fixAll.eslint": **true**

}

}

이렇게 ESLint 익스텐션으로는 실시간 코드 품질 검사를 하고 저장시 자동 포메팅을 하도록 하면 실시간으로 코드 품질을 검사하고 포맷도 일관적으로 유지할 수 있다.

## 5. 정리

읽기 좋은 코드는 유지보수 하기좋다. 그만큼 어플리케이션의 수명은 오래갈 수 있다. 여럿이서 함께 일하는 환경에서 손으로 코드를 관리하는 것은 무척 번거럽고 어쩌면 불가능한 일일지도 모른다. 규칙이 정해졌고 자동화할 수 있다면 도구의 도움을 받는 것이 현명하다.

ESLint는 오류와 버그의 가능성을 찾아 코드 품질을 높이는 역할을 한다. 프리티어는 코드를 일관적으로 포매팅하기 때문에 읽기 수월한 코드를 만들어 준다. 이러한 도구를 개발 플로우의 적절한 시점에 통합하여 자동화하면 개발자는 좀 더 본질적인 코딩에 집중할 수 있을 것이다.

# 프론트엔드 개발환경의 이해: 웹팩(심화)

2020년 01월 02일

이전글 [웹팩(기본편)](http://jeonghwan-kim.github.io/series/2019/12/10/frontend-dev-env-webpack-basic.html)에서는 웹팩의 개념과 간단한 사용법에 대해 살펴보았다. 웹팩은 프론트엔드 개발 서버를 제공하고, 몇 가지 방법으로 빌드 결과를 최적화 할 수 있는데 이번 글에서 자세히 살펴 보겠다.

## 1. 웹팩 개발 서버

### 1.1 배경

지금까지는 브라우져에 파일을 직접 로딩해서 결과물을 확인했다. 인터넷에 웹사이트를 게시하려면 서버 프로그램으로 이 파일을 읽고 요청한 클라이언트에게 제공해야 한다.

개발환경에서도 이와 유사한 환경을 갖추는 것이 좋다. 운영환경과 맞춤으로써 배포시 잠재적 문제를 미리 확인할 수 있다. 게다가 ajax 방식의 api 연동은 cors 정책 때문에 반드시 서버가 필요하다.

프론트엔드 개발환경에서 이러한 개발용 서버를 제공해 주는 것이 [webpack-dev-server](https://webpack.js.org/configuration/dev-server/)다.

### 1.2 설치 및 사용

먼저 webpack-dev-server 패키지를 설치한다.

npm i -D webpack-dev-server

node\_modules/.bin에 있는 webpack-dev-servr 명령어를 바로 실행해도 되지만 npm 스크립트로 등록해서 사용하겠다.

package.json:

{

"scripts": {

"start": "webpack-dev-server"

}

}

npm start 명령어로 실행하면 다음과 같이 서버가 구동되었다는 메시지를 확인할 수 있다.

npm start

**>** webpack-dev-server

ℹ ｢wds｣: Project is running at http://localhost:8080/

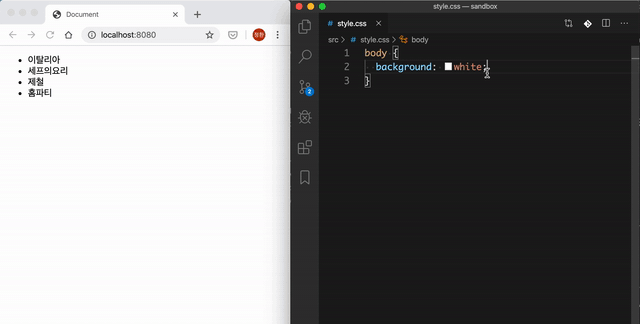
ℹ ｢wds｣: webpack output is served from /

ℹ ｢wds｣: Content not from webpack is served from

로컬 호스트의 8080 포트에 서버가 구동되어서 접속을 대기하고 있다. 웹팩 아웃풋인 dist 폴더는 루트 경로를 통해 접속할 수 있다.

브라우져 주소창에 http://localhost:8080 으로 접속해 보면 결과물을 확인할 수 있다.

소스 코드를 수정하고 저장해 보자. 웹팩 서버는 파일 변화를 감지하면 웹팩 빌드를 다시 수행하고 브라우져를 리프레시하여 변경된 결과물을 보여준다.



이것만으로도 개발 환경이 무척 편리해졌다. 코드를 수정할때마다 저장하고 브라우져 갱신 버튼을 클릭하는 것은 무척 지난한 일인데 말이다.

### 1.3 기본 설정

웹팩 설정 파일의 devServer 객체에 개발 서버 옵션을 설정할 수 있다.

*// webpack.config.js:*

module.exports **=** {

devServer: {

contentBase: path.join(\_\_dirname, "dist"),

publicPath: "/",

host: "dev.domain.com",

overlay: **true**,

port: 8081,

stats: "errors-only",

historyApiFallback: **true**,

}

}

**contentBase**: 정적파일을 제공할 경로. 기본값은 웹팩 아웃풋이다.

**publicPath**: 브라우져를 통해 접근하는 경로. 기본값은 ‘/’ 이다.

**host**: 개발환경에서 도메인을 맞추어야 하는 상황에서 사용한다. 예를들어 쿠기 기반의 인증은 인증 서버와 동일한 도메인으로 개발환경을 맞추어야 한다. 운영체제의 호스트 파일에 해당 도메인과 127.0.0.1 연결한 추가한 뒤 host 속성에 도메인을 설정해서 사용한다.

**overlay**: 빌드시 에러나 경고를 브라우져 화면에 표시한다.

**port**: 개발 서버 포트 번호를 설정한다. 기본값은 8080.

**stats**: 메시지 수준을 정할수 있다. ‘none’, ‘errors-only’, ‘minimal’, ‘normal’, ‘verbose’ 로 메세지 수준을 조절한다.

**historyApiFallBack**: 히스토리 API를 사용하는 SPA 개발시 설정한다. 404가 발생하면 index.html로 리다이렉트한다.

이 외에도 개발 서버를 실행할때 명령어 인자로 --progress를 추가하면 빌드 진행율을 보여준다. 빌드 시간이 길어질 경우 사용하면 좋다.

메세지 출력 옵션만 설정한 뒤,

*// webpack.config.js:*

module.exports **=** {

devServer: {

overlay: **true**,

stats: 'errors-only',

}

}

npm 스크립트를 수정해보자.

{

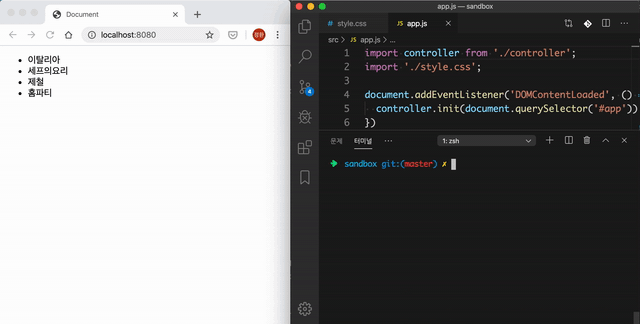
"scripts": {

"start": "webpack-dev-server --progress"

}

}

빌드하면 아래와 같이 출력 메세지를 조절할 수 있다.



이외에도 다양한 옵션은 [여기](https://webpack.js.org/configuration/dev-server/)를 참고하자.

## 2. API 연동

프론트엔드에서는 서버와 데이터 주고 받기 위해 ajax를 사용한다. 보통은 api 서버를 어딘가에 띄우고(혹은 로컬호스트에 띄우고) 프론트 서버와 함께 개발한다. 개발 환경에서 이러한 api 서버 구성을 어떻게 하는지 알아 보자.

### 2.1 목업 API 1: devServer.before

웹팩 개발 서버 설정 중 [before](https://webpack.js.org/configuration/dev-server/#devserverbefore) 속성은 웹팩 서버에 기능을 추가할 수 있는 여지를 제공한다. 이것을 이해하려면 노드 Express.js에 사전지식이 있으면 유리한데, 간단히 말하면 익스프레스는 미들웨어 형태로 서버 기능을 확장할 수 있는 웹프레임웍이다. devServer.before에 추가하는 것이 바로 미들웨어인 셈이다. 아래 코드를 보자.

*// webpack.config.js*

module.exports **=** {

devServer: {

before: (app, server, compiler) **=>** {

app.**get**('/api/keywords', (req, res) **=>** {

res.json([

{ keyword: '이탈리아' },

{ keyword: '세프의요리' },

{ keyword: '제철' },

{ keyword: '홈파티'}

])

})

}

}

}

before에 설정한 미들웨어는 익스프레스에 의해서 app 객체가 인자로 전달되는데 Express 인스턴스다. 이 객체에 라우트 컨트롤러를 추가할 수 있는데 app.get(url, controller) 형태로 함수를 작성한다. 컨트롤러에서는 요청 req과 응답 res 객체를 받는데 여기서는 res.json() 함수로 응답하는 코드를 만들었다.

웹팩 개발 서버는 GET /api/keywords 요청시 4개의 엔트리를 담은 배열을 반환할 것이다. 서버를 다시 구동하고 curl로 http 요청을 보내보자.

curl localhost:8080/api/keywords

**[{**"keyword":"이탈리아"**}**,**{**"keyword":"세프의요리"**}**,**{**"keyword":"제철"**}**,**{**"keyword":"홈파티"**}]**

이런 기능이 왜 필요할까? 개발 초기 서버 api가 만들어지기 전, 서버 api 응답을 프론트엔드에서 추가할 때 사용할 수 있다. 익스프레스 사전 지식이 있다면 여기에 다양한 서버 응답을 구현할 수 있다.

프론트 코드를 수정해서 방금 만든 엔드폰인트를 호출하는 코드로 변경해 보자. ajax 라이브러리인 axios를 설치한다.

npm i axios

프론트엔드의 model.js 코드를 다음과 같이 수정한다.

*// src/model.js:*

**import** axios **from** 'axios'

*// const data = [*

*// {keyword: '이탈리아'},*

*// {keyword: '세프의요리'},*

*// {keyword: '제철'},*

*// {keyword: '홈파티'},*

*// ]*

**const** model **=** {

**async** **get**() {

*// return data*

**const** result **=** **await** axios.**get**('/api/keywords');

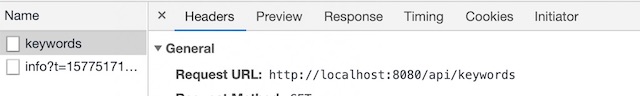
**return** result.data;

}

}

**export** **default** model;

기존에는 data에 데이터를 관리했는데 이제는 ajax 호출 후 응답된 데이터를 반환하도록 변경했다. 화면을 확인해 보면 웹펙에서 설정한 api 응답이 화면에 나오는걸 확인할 수 있다.



### 2.2 목업 API 2: connect-api-mocker

목업 api 작업이 많을때는 [connect-api-mocker](https://github.com/muratcorlu/connect-api-mocker) 패키지의 도움을 받자. 특정 목업 폴더를 만들어 api 응답을 담은 파일을 저장한 뒤, 이 폴더를 api로 제공해 주는 기능을 한다.

먼저 이 패키지를 설치하고,

npm i -D connect-api-mocker

mocks/api/keywords/GET.json 경로에 API 응답 파일을 만든다.

GET 메소드를 사용하기때문에 GET.json으로 파일을 만들었다(물론 POST, PUT, DELETE 도 지원).

GET.json:

[

{ "keyword": "이탈리아" },

{ "keyword": "세프의요리" },

{ "keyword": "제철" },

{ "keyword": "홈파티 " }

]

기존에 설정한 목업 응답 컨트롤러를 제거하고 connect-api-mocker로 미들웨어를 대신한다.

*// webpack.config.js:*

**const** apiMocker **=** require('connect-api-mocker')

module.exports **=** {

devServer: {

before: (app, server, compiler) **=>** {

app.use(apiMocker('/api', 'mocks/api'))

},

}

}

익스프레스 객체인 app은 get() 메소드 뿐만 아니라 미들웨어 추가를 위한 범용 메소드 use()를 제공하는데, 이를 이용해 목업 미들웨어를 추가했다. 첫번째 인자는 설정할 라우팅 경로인데 /api로 들어온 요청에 대해 처리하겠다는 의미다. 두번째 인자는 응답으로 제공할 목업 파일 경로인데 방금 만든 mocks/api 경로를 전달했다.

목업 API 갯수가 많다면 직접 컨트롤러를 작성하는 것 보다 목업 파일로 관리하는 것을 추천한다.

### 2.3 실제 API 연동: devServer.proxy

이번에는 api 서버를 로컬환경에서 띄운 다음 목업이 아닌 이 서버에 직접 api 요청을 해보자. 로컬호스트 8081 포트에 아래와 같이 서버가 구성되었다고 가정하겠다.

$ curl localhost:8081/api/keywords

**[{**"keyword":"이탈리아"**}**,**{**"keyword":"세프의요리"**}**,**{**"keyword":"제철"**}**,**{**"keyword":"홈파티"**}]**

ajax 요청 부분의 코드를 변경한다.

*// src/model.js*

**const** model **=** {

**async** **get**() {

*// const result = await axios.get('/api/keywords');*

*// 직접 api 서버로 요청한다.*

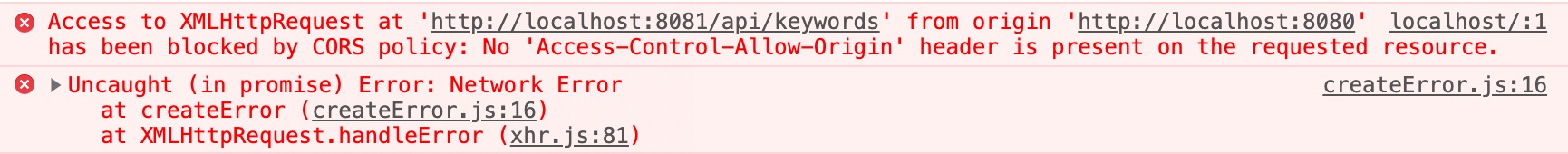
**const** { data } **=** **await** axios.**get**('http://localhost:8081/api/keywords');

**return** data;

}

}

웹팩 개발서버를 띄우고 화면을 확인해 보자. 잘 나오는가? 브라우져 개발자 도구에 보면 다음과 같은 오류 메세지가 출력된다.



localhost:8080에서 localhost:8081 로 ajax 호출을 하지 못하는데 이유는 CORS 정책 때문이라는 메세지다. 요청하는 리소스에 “Access-Control-Allow-Origin” 헤더가 없다는 말도 한다.

[CORS(Cross Origin Resource Shaing)](https://developer.mozilla.org/ko/docs/Web/HTTP/Access_control_CORS이란) 브라우져와 서버간의 보안상의 정책인데 브라우저가 최초로 접속한 서버에서만 ajax 요청을 할 수 있다는 내용이다. 방금같은 경우는 localhost로 같은 도메인이지만 포트번호가 8080, 8081로 달라서 다른 서버로 인식하는 것이다.

해결하는 방법은 두 가지인데 먼저 서버측 솔루션 부터 보자. 해당 api 응답 헤더에 “Access-Control-Allow-Origiin: \*” 헤더를 추가한 뒤 응답하면, 브라우져에서 응답 데이터를 받을 수 있다.

*// server/index.js*

app.**get**('/api/keywords', (req, res) **=>** {

res.header("Access-Control-Allow-Origin", "\*"); *// 헤더를 추가한다*

res.json(keywords)

})

한편 프론트엔드 측 해결방법을 보자. 서버 응답 헤더를 추가할 필요없이 웹팩 개발 서버에서 api 서버로 **프록싱**하는 것이다. 웹팩 개발 서버는 [proxy](https://webpack.js.org/configuration/dev-server/#devserverproxy) 속성으로 이를 지원한다.

*// webpack.config.js*

module.exports **=** {

devServer: {

proxy: {

'/api': 'http://localhost:8081', *// 프록시*

}

}

}

개발서버에 들어온 모든 http 요청중 /api로 시작되는것은 http://localhost:8081로 요청하는 설정이다. api 호출코드를 다시 복구한 뒤,

*// src/model.js*

**const** model **=** {

**async** **get**() {

*// const { data } = await axios.get('http://localhost:8081/api/keywords');*

**const** { data } **=** **await** axios.**get**('/api/keywords');

**return** data;

}

}

확인해보면 정상 동작하는 것을 확인할 수 있다.

## 3. 핫 모듈 리플레이스먼트

### 3.1 배경

웹팩 개발서버는 코드의 변화를 감지해서 전체 화면을 갱신하기 때문에 개발 속도를 높일 수 있다. 하지만 어떤 상황에서는 전체 화면을 갱신하는 것이 좀 불편한 경우도 있다. 싱글페이지어플리케이션은 브라우져에서 데이터를 들고 있기 때문에 리프레시 후에 모든 데이터가 초기화 되어 버리기 때문이다. 다른 부분을 수정했는데 입력한 폼 데이터가 날아가 버리는 경우도 있고 말이다.

전체 화면 갱신 하지 않고 변경한 모듈만 바꿔치기 한다면 어떨까? **핫 모듈 리플레이스먼트**는 이러한 목적으로 제공되는 웹팩 개발서버의 한 기능이다.

### 3.2 설정

설정은 간단하다. [devServer.hot](https://webpack.js.org/configuration/dev-server/#devserverhot) 속성을 켠다.

*// webpack.config.js:*

module.exports **=** {

devServer **=** {

hot: **true**,

},

}

view.js를 사용하는 컨트롤러 코드를 잠깐 읽어보자.

*// src/controller.js*

**import** model **from** "./model";

**import** view **from** "./view";

**const** controller **=** {

**async** init(el) {

**this**.el **=** el

view.render(**await** model.**get**(), **this**.el);

}

}

**export** **default** controller;

컨트롤러는 model과 view에 의존성이 있는데 이 둘을 이용해 데이터를 가져와 화면을 렌더한다. 만약 view 모듈에 변화가 있을 경우 전체 화면을 갱신하지 않고 변경된 view 모듈만 다시 실행하는 것이 핫 모듈의 작동 방식이다.

이 기능을 만들기 위해 컨트롤러 하단에 다음 코드를 추가해 보자.

*// src/controller.js*

*// 중략*

**export** **default** controller;

**if** (module.hot) {

console.log('핫모듈 켜짐')

module.hot.accept('./view', () **=>** {

console.log('view 모듈 변경됨')

})

}

devServer.hot 옵션을 켜면 웹팩 개발 서버 위에서 module.hot 객체가 생성된다. 이 객체의 accept() 메소드는 감시할 모듈과 콜백 함수를 인자로 받는다. 위에서는 view.js 모듈을 감시하고 변경이 있으면 전달한 콜백 함수가 동작하도록 했다.

웹팩 개발 서버를 재 시작하면 브라우져에 다음과 같이 로그가 찍힌다.



후에 view.js 파일을 수정하면 다음 로그가 찍히는 것을 확인할 수 있다.



이 콜백 함수 안에서 변경된 view 모듈을 이용하면 view 모듈을 교체할 수 있을 것 같다. model로 데이터를 부르고 다시 변경된 view 모듈로 렌더 함수를 실행했다.

*// src/controller.js*

**if** (module.hot) {

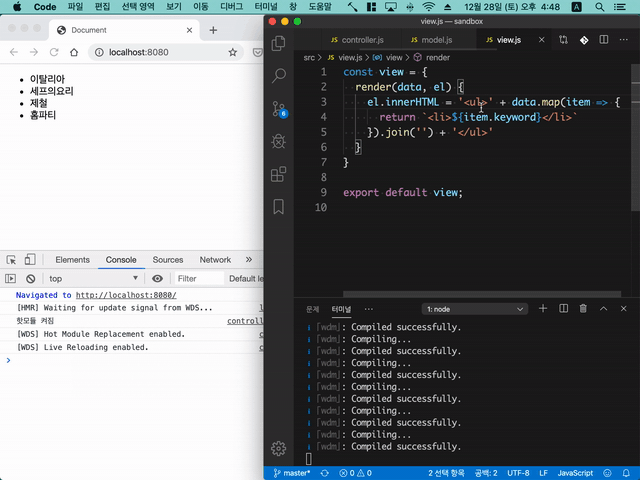
module.hot.accept('./view', **async** () **=>** {

view.render(**await** model.**get**(), controller.el); *// 변경된 모듈로 교체*

})

}

view.js 코드를 변경하고 저장하면 브라우져 갱신 없이 화면이 변경된다.



### 3.3 핫로딩을 지원하는 로더

이러한 HMR 인터페이스를 구현한 로더만이 핫 로딩을 지원하는데 웹팩 기본편에서 보았던 style-loader가 그렇다. 잠깐 코드를 보면 hot.accept() 함수를 사용한 것을 알 수 있다.



참고: [style-loader 코드](https://github.com/webpack-contrib/style-loader/blob/master/src/index.js#L34-L37)

이 외에도 리액트를 지원하는 react-hot-loader, 파일을 지원하는 file-loader는 핫 모듈 리플레이스먼트를 지원하는데 [여기](https://webpack.js.org/guides/hot-module-replacement/#other-code-and-frameworks)를 참고하자.

## 4. 최적화

코드가 많아지면 번들링된 결과물도 커지기 마련이다. 거의 메가바이트 단위로 커질수도 있는데 브라우져 성능에 영향을 줄 수 있다. 파일을 다운로드하는데 시간이 많이 걸리기 때문이다. 이번 섹션에서는 번들링한 결과물을 어떻게 최적화 할수 있는지 몇가지 방법에 대해 알아보겠다.

### 4.1 production 모드

웹팩에 내장되어 있는 최적화 방법중 [mode](https://webpack.js.org/configuration/mode/) 값을 설정하는 방식이 가장 기본이다. 세 가지 값이 올 수 있는데 지금까지 설정한 “development”는 디버깅 편의를 위해 아래 두 개 플러그인을 사용한다.

* NamedChunksPlugin
* NamedModulesPlugin

DefinePlugin을 사용한다면 process.env.NODE\_ENV 값이 “development”로 설정되어 어플리케이션에 전역변수로 주입된다.

반면 mode를 “production”으로 설정하면 자바스크립트 결과물을 최소화 하기 위해 다음 일곱 개 플러그인을 사용한다.

* FlagDependencyUsagePlugin
* FlagIncludedChunksPlugin
* ModuleConcatenationPlugin
* NoEmitOnErrorsPlugin
* OccurrenceOrderPlugin
* SideEffectsFlagPlugin
* TerserPlugin

DefinePlugin을 사용한다면 process.env.NODE\_ENV 값이 “production” 으로 설정되어 어플리케이션 전역변수로 들어간다.

그럼 환경변수 NODE\_ENV 값에 따라 모드를 설정하도록 웹팩 설정 코드를 다음과 같이 추가할 수 있겠다.

*// webpack.config.js:*

**const** mode **=** process.env.NODE\_ENV **||** 'development'; *// 기본값을 development로 설정*

module.exports **=** {

mode,

}

빌드 시에 이를 운영 모드로 설정하여 실행하도록 npm 스크립트를 추가한다.

package.json:

{

"scripts": {

"start": "webpack-dev-server --progress",

"build": "NODE\_ENV=production webpack --progress"

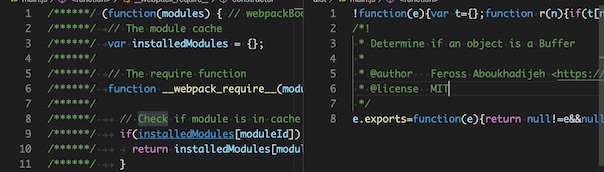
}

}

start는 개발 서버를 구동하기 때문에 환경변수를 설정하지 않고 기본값 development를 사용할 것이다. 배포용으로 만들 build는 환경변수를 production으로 설정했고 웹팩 mode에 설정된다.

빌드한 뒤 결과물을 확인해 보자.

npm run build



왼쪽에 development로 설정해서 빌드한 결과물과 비교해 보면 오른쪽에 production으로 빌드한 결과물의 확연한 차이를 볼 수 있다.

### 4.2 optimazation 속성으로 최적화

빌드 과정을 커스터마지징할 수 있는 여지를 제공하는데 그것이 바로 [optimazation](https://webpack.js.org/configuration/optimization/) 속성이다.

HtmlWebpackPlugin이 html 파일을 압축한것 처럼 css 파일도 빈칸을 없애는 압축을 하려면 어떻게 해야할까? [optimize-css-assets-webpack-plugin](https://webpack.js.org/plugins/mini-css-extract-plugin/#minimizing-for-production)이 바로 그것이다.

플러그인을 다운로드 하고,

npm i -D optimize-css-assets-webpack-plugin

웹팩 설정을 추가한다.

*// webpack.config.js:*

**const** OptimizeCSSAssetsPlugin **=** require('optimize-css-assets-webpack-plugin');

module.exports **=** {

optimization: {

minimizer: mode **===** 'production' ? [

**new** OptimizeCSSAssetsPlugin(),

] : [],

},

}

[optimization.minimizer](https://webpack.js.org/configuration/optimization/#optimizationminimizer)는 웹팩이 결과물을 압축할때 사용할 플러그인을 넣는 배열이다. 설치한 OptimizeCSSAssetsPlugin을 전달해서 빌드 결과물중 css 파일을 압축하도록 했다.

빌드하뒤 확인하면 css 파일도 압축되었다.

css 빌드 결과

mode=production일 경우 사용되는 [TerserWebpackPlugin](https://webpack.js.org/plugins/terser-webpack-plugin/)은 자바스크립트 코드를 난독화하고 debugger 구문을 제거한다. 기본 설정 외에도 [콘솔 로그를 제거하는 옵션](https://github.com/terser/terser#compress-options)도 있는데 배포 버전에는 로그를 감추는 것이 좋을 수도 있기 때문이다.

이 플러그인을 설치한 뒤,

npm i -D terser-webpack-plugin

optionmization.minimizer 배열에 추가한다.

*// webpack.config.js:*

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

module.exports **=** {

optimization: {

minimizer: mode **===** 'production' ? [

**new** TerserPlugin({

terserOptions: {

compress: {

drop\_console: **true**, *// 콘솔 로그를 제거한다*

}

}

}),

] : [],

},

}

### 4.3 코드 스플리팅

코드를 압축하는 것 외에도 아예 결과물을 여러개로 쪼개면 좀 더 브라우져 다운로드 속도를 높일 수 있다. 큰 파일 하나를 다운로드 하는것 보다 작은 파일 여러개를 동시에 다운로드하는 것이 더 빠르기 때문이다.

가장 단순한 것은 엔트리를 여러개로 분리하는 것이다.

*// webpack.config.js*

module.exports **=** {

entry: {

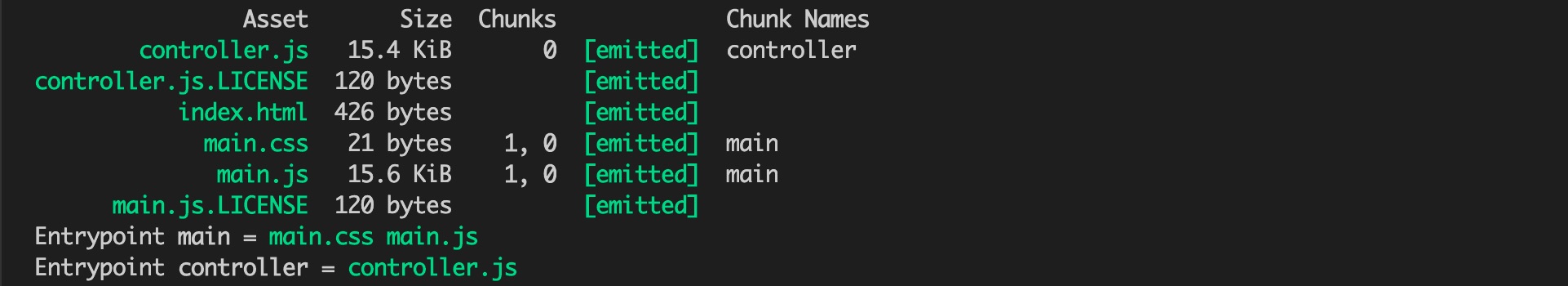
main: "./src/app.js",

controller: "./src/controller.js",

}

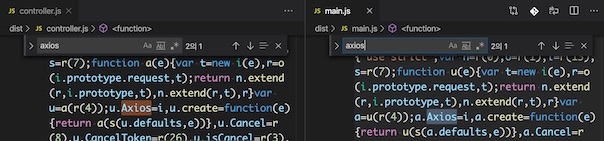
}

빌드하면 엔트리가 두 개 생성되고 물론 하나의 엔트이일 때보다 용량이 조금 줄었다.



모듈을 어떻게 분리하는냐에 따라 이 결과물의 크기를 조절할 수 있는데 지금은 거의 변화가 없다. HtmlWebpackPlugin에 의해 html 코드에소 두 파일을 로딩하는 코드도 추가된다.

하지만 두 파일을 비교해 보면 중복코드가 있다.



axios 모듈인데 main, controller 둘 다 axios를 사용하기 때문이다.

[SplitChunksPlugin](https://webpack.js.org/guides/code-splitting/#prevent-duplication)은 코드를 분리할때 중복을 예방하는 플러그인이다. optization.splitChucks 속성을 설정하는 방식이다.

*// webpack.config.js:*

module.exports **=** {

optimization: {

splitChunks: {

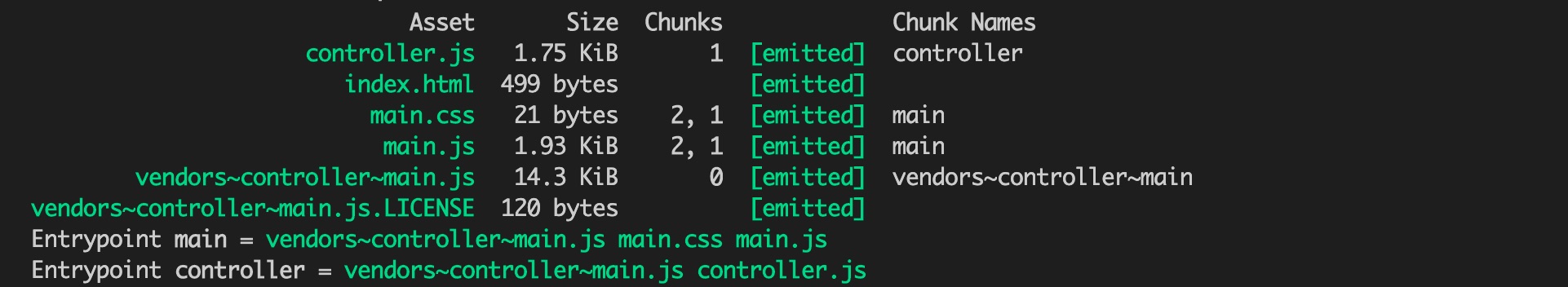
chunks: 'all',

},

},

}

다시 빌드해보자.



main.js, controller.js외에도 vendors~controller\_main.js 파일도 생겼다. 마지막 파일은 두 엔트리의 중복 코드를 담은 파일이다. axios로 검색하면 main.js와 controller.js에서는 없고 vendors~controller~main.js에만 있다.

이런 방식은 엔트리 포인트를 적절히 분리해야기 때문에 손이 많이 가는 편이다. 반면 자동으로 변경해 주는 방식이 있는데 이를 **다이나믹 임포트**라고 부른다.

기존 컨트롤러 코드를 보면 이렇다.

**import** controller **from** './controller';

document.addEventListener('DOMContentLoaded', () **=>** {

controller.init(document.querySelector('#app'))

})

import/from으로 컨트롤러 모듈을 가져와서 사용했다.

이를 동적으로 임포트하려면 다음처럼 변경한다.

**function** getController() {

**return** **import**(*/\* webpackChunkName: "controller" \*/* './controller').then(m**=>** {

**return** m.**default**

})

}

document.addEventListener('DOMContentLoaded', () **=>** {

getController().then(controller **=>** {

controller.init(document.querySelector('#app'))

})

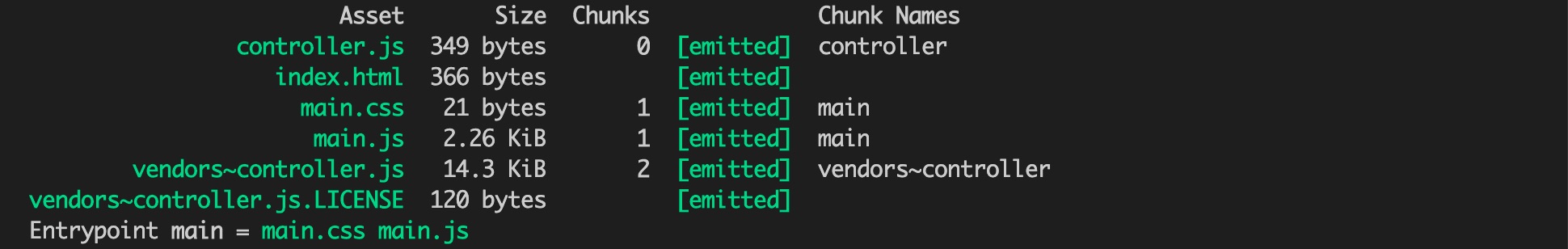
})

getController() 함수를 정의했는데 컨트롤러 모듈을 가져는 함수다. import() 함수로 가져올 컨트롤러 모듈 경로를 전달하는데 주석으로 webpackHunkName: “controller”를 전달했다. 이것은 웹펙이 이 파일을 처리할때 청크로 분리하는데 그 청그 이름을 설정한 것이다.

그리고 나서 프라미스를 반환하는 getController() 함수로 모듈을 가져와 사용하였다.

변경한 웹팩 설정 파일도 다시 복구해야 한다. 엔트리 포인트를 다시 main만 남겨두고 optimization에 설정한 SplitChunksPlugin 옵션도 제거한다.

빌드하면 자동으로 파일이 분리되었다.



엔트리를 분리하지 않아도 controller와 app의 중복코드를 vendors~controller.js 파일로 분리한다. 다이나믹 임포트로 모듈을 가져오면 단일 엔트리를 유지하면서 코드를 분리할 수 있다.

### 4.4 externals

조금만 더 생각해 보면 최적화해 볼 수 있는 부분이 있다. 바로 axios같은 써드파티 라이브러리다. 패키지로 제공될때 이미 빌드 과정을 거쳤기 때문에 빌드 프로세스에서 제외하는 것이 좋다. 웹팩 설정중 [externals](https://webpack.js.org/configuration/externals/)가 바로 이러한 기능을 제공한다.

*// webpack.config.js:*

module.exports **=** {

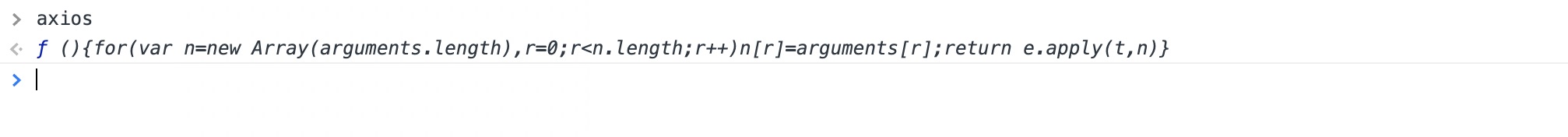
externals: {

axios: 'axios',

},

}

externals에 추가하면 웹팩은 코드에서 axios를 사용하더라도 번들에 포함하지 않고 빌드한다. 대신 이를 전역 변수로 접근하도록하는데 키로 설정한 axios가 그 이름이다.



axios는 이미 node\_modules에 위치해 있기 때문에 이를 웹팩 아웃풋 폴더에 옮기고 index.html에서 로딩해야한다. 파일을 복사하는 [CopyWebpackPlugin](https://webpack.js.org/plugins/copy-webpack-plugin/)을 설치한다.

npm i -D copy-webpack-plugin

플러그인을 사용해서 라이브러리를 복사한다.

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

module.exports **=** {

plugins: [

**new** CopyPlugin([{

from: './node\_modules/axios/dist/axios.min.js',

to: './axios.min.js' *// 목적지 파일에 들어간다*

}])

]

}

마지막으로 index.html에서는 axios를 로딩하는 코드를 추가한다.

*<!-- src/index.html -->*

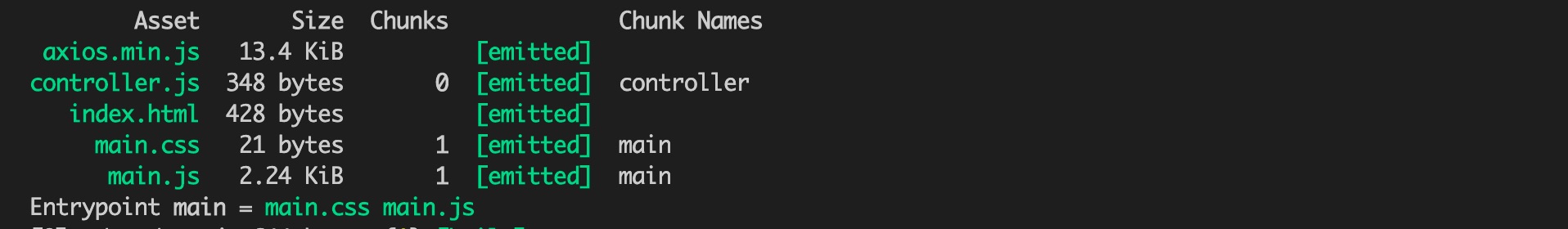
<script type="text/javascript" src="axios.min.js"></script>

</body>

</html>

axios는 이렇게 직접 추가했지만 번들링한 결과물은 htmlwebpacPlugin이 주입해 주는 것을 잊지말자.

다시 빌드해 보면……



axios는 빌드하지 않고 복사만 한다. controller와 main이 분리되었다. 이전에는 공통의 코드인 axios가 vender~.js로 분리되었는데 지금은 파일조차 없다. 만약 써드파티 라이브러리 외에 공통의 코드가 있다면 이 파일로 분리되었을 것이다.

이렇게 써드파티 라이브러리를 externals로 분리하면 용량이 감소뿐만 아니라 빌드시간도 줄어들고 덩달아 개발 환경도 가벼워질 수 있다.

## 5. 정리

웹팩 사용방법에 대해 좀더 알아 보았다.

개발 서버를 띄워 파일 감지, api 서버 연동 등 개발 환경을 좀 더 편리하게 구성할 수 있었다. 특히 핫 모듈 리플레이스먼트는 일부 모듈의 변경만 감지하여 페이지 갱신 없이 변경사항을 브라우져에 렌더링할 수 있다.

웹팩 최적화 방법에 대해서도 알아보았다. mode 옵션을 production으로 설정하면 웹팩 내장 플러그인이 프로덕션 모드로 동작한다. 번들링 결과물 크기가 커지면 브라우져에서 다운로딩하는 성능이 떨어질수 있는데 코드 스플리트 기법을 사용해서 해결할 수 있다. 써드파티 라이브러리는 externals로 옮겨 빌드 과정에서 제외할수 있다.