

# Webpack - Module Bundler

# 개요

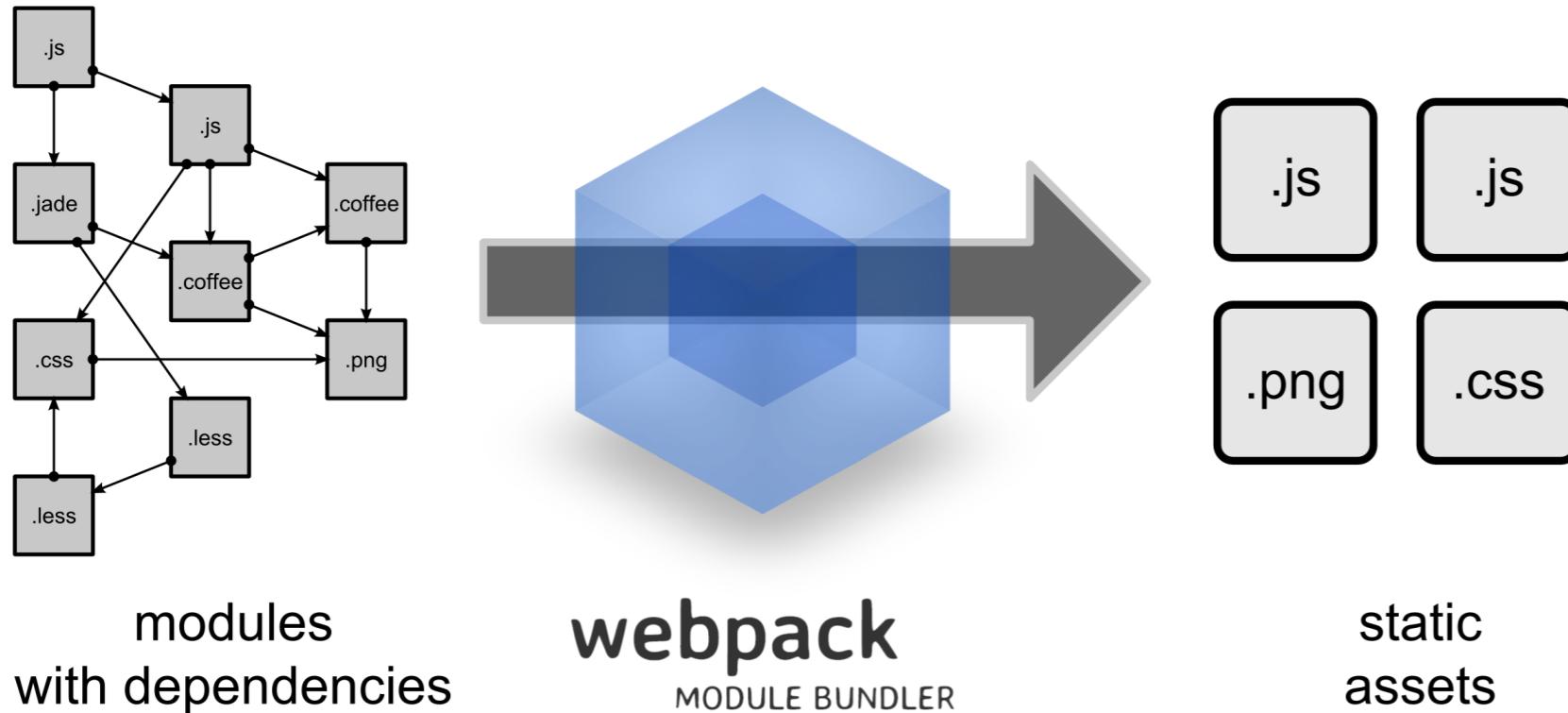
- React, Angular, Vue에서 권고하는 Webpack 설정에 대해 학습 및 이해
- Webpack의 주요 설정 Entry, Output, Loader, Plugins, Resolve 학습 및 실습
- 실제 개발환경에서 사용할 수 있는 Webpack 개발환경 설정방법 학습 및 실습

# 목차

- Webpack 소개 및 배경
- Webpack 사용에 필요한 cli 와 npm 명령어
- Webpack Getting started (실습)
- Webpack Entry
- Webpack Output
- Webpack Loader (실습)
- Webpack Plugins (실습)
- Webpack 개발환경 세팅 (실습)
- Webpack Resolve (실습)
- 기타

# Webpack 은 무엇인가?

- 서로 연관 관계가 있는 웹 자원들을 js, css, img 와 같은 스태틱한 자원으로 변환해주는 모듈 번들러
- 아래 사진은 직관적으로 webpack 의 역할을 설명



# Webpack 을 사용하는 이유 & 배경?

## 1. 새로운 형태의 Web Task Manager

- 기존 Web Task Manager (Gulp, Grunt) 의 기능 + 모듈 의존성 관리
- 예) minification 을 webpack default cli 로 실행 가능

```
webpack -p
```

## 2. 자바스크립트 Code based Modules 관리

- 기존 모듈 로더들과의 차이점 : 모듈 간의 관계를 청크 (chunk) 단위로 나눠 필요할 때 로딩
- 현대의 웹에서 JS 역할이 커짐에 따라, Client Side 에 들어가는 코드량이 많아지고 복잡해짐
- 복잡한 웹 앱을 관리하기 위해 모듈 단위로 관리하는 Common JS, AMD, ES6 Modules 등이 등장
- 모듈 관리에 대해 `script` 태그로 JS 를 모듈화 하는 간단한 예제

```
<script src="module1.js"></script>
<script src="module2.js"></script>
<script src="library1.js"></script>
<script src="module3.js"></script>
```

- 상기 모듈 로딩 방식의 문제점 : 전역변수 충돌, 스크립트 로딩 순서, 복잡도에 따른 관리상의 문제
- 이를 해결하기 위해 AMD 및 기타 모듈 로더들이 등장.
- 가독성이나 다수 모듈 미병행 처리등의 약점을 보완하기 위해 Webpack 이 등장

# Webpack 의 철학

## 1. Everything is Module

모든 웹 자원 (js, css, html) 이 모듈 형태로 로딩 가능

```
require('base.css');
require('main.js');
```

## 2. Load only “what” you need and “when” you need

초기에 불필요한 것들을 모두 로딩하지 않고, 필요할 때 필요한 것만 로딩하여 사용

## Webpack 에 필요한 NPM 명령어

- `npm init` 웹팩 초기 설정에 필요한 명령어로 package.json 파일을 생성
- `npm install (i)` 라이브러리 명 (여러개 한번에 가능)

```
npm i jquery angular lodash --save
```

# Webpack 명령어

- `webpack` : 웹팩 빌드 기본 명령어 (주로 개발용)
- `webpack -p` : minification 기능이 들어간 빌드 (주로 배포용)
- `webpack -watch(-w)` : 개발에서 빌드할 파일의 변화를 감지
- `webpack -d` : sourcemap 포함하여 빌드
- `webpack --display-error-details` : error 발생시 디버깅 정보를 상세히 출력
- `webpack --optimize-minimize --define process.env.NODE_ENV='production'` : 배포용

# Webpack Getting Started

간단한 webpack 기본 실습

## 실습절차

1. webpack 전역 설치
2. `npm init` 으로 package.json 생성
3. app/index.js 와 index.html 생성
4. js 와 html 에 코드 추가
5. `webpack app/index.js dist/bundle.js` 명령어 실행 후 index.html 로딩
6. webpack.config.js 파일 추가 후 webpack

# Webpack Entry

- webpack 으로 묶은 모든 라이브러리들을 로딩할 시작점 설정
- a,b,c 라이브러리를 모두 번들링한 bundle.js 를 로딩한다
- 1개 또는 2개 이상의 엔트리 포인트를 설정할 수 있다.

```
1 {  
2     entry: './public/src/index.js',  
3  
4     output: {  
5         path: '/dist',  
6         filename: "bundle.js"  
7     }  
8 }  
9  
10 }  
  
2 {  
3     entry: ['./public/src/index.js'],  
4  
5     output: {  
6         path: '/dist',  
7         filename: "bundle.js"  
8     }  
9 }  
10  
11 }  
12 }  
  
2 {  
3     entry: {  
4         index: './public/src/index.js'  
5     },  
6  
7     output: {  
8         path: '/dist',  
9         filename: "bundle.js"  
10    }  
11 }  
12 }
```

## 여러가지 Entry 유형

```
var config = {
  // #1 - 간단한 entry 설정
  entry: './path/to/my/entry/file.js'
  // #2 - 앱 로직용, 외부 라이브러리용
  entry: {
    app: './src/app.js',
    vendors: './src/vendors.js'
  }
  // #3 - 페이지당 불러오는 js 설정
  entry: {
    pageOne: './src/pageOne/index.js',
    pageTwo: './src/pageTwo/index.js',
    pageThree: './src/pageThree/index.js'
  }
};
```

# Mutiple Entry points

- 앞에 복수개 엔트리 포인트에 대한 설정 예시

```
// webpack.config.js
module.exports = {
  entry: {
    Profile: './profile.js',
    Feed: './feed.js'
  },
  output: {
    path: 'build',
    filename: '[name].js' // 위에 지정한 entry 키의 이름에 맞춰서 결과 산출
  }
};

// 번들파일 Profile.js 를 <script src="build/Profile.js"></script> 로 HTML 에 삽입
```

## CommonsChunkPlugin

- 여러개의 엔트리 포인트에 주입되는 공통의 모듈을 관리하기 위한 플러그인
- 모든 엔트리 포인트에 주입되는 라이브러리를 효율적으로 관리하기 위해, Chunk라는 단위로 미리 분리하여 관리한다.
- 모든 번들에 로딩될 필요 없이, 초기에 한번만 로딩하는 장점과 캐쉬에 용이하다는 장점이 있다.

# Webpack Output

- entry에서 설정하고 묶은 파일의 결과값을 설정

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

## path vs public Path

- webpack dev server 의 path, publicPath 를 구분하기 위해 파악
- output 의 path 와 public path 속성의 차이점 이해 필요

```
# Webpack 컴파일 시에 뜨는 로그
Project is running at http://localhost:9000/
webpack output is served from /dist/
```

- `output.path` : 번들링한 결과가 위치할 번들링 파일의 절대 경로
- `output.publicPath` : 브라우저가 참고할 번들링 결과 파일의 URL 주소를 지정. (CDN 을 사용하는 경우 CDN 호스트 지정)

```
// publicPath 예제 #1
output: {
  path: "/home/proj/public/assets",
  publicPath: "/assets/"
}

// publicPath 예제 #2
output: {
  path: "/home/proj/cdn/assets/[hash]",
  publicPath: "http://cdn.example.com/assets/[hash]/"
}
```

## CDN에 이미지를 올려놓은 경우

```
// webpack.config.js
output: {
  publicPath: 'https://cdn.example.com/assets/'
}
```

```
<!-- 템플릿 -->

```

# Webpack Loader

webpack can only process JavaScript natively, but loaders are used to transform other resources into JavaScript. By doing so, every resource forms a module.

- 웹팩은 자바스크립트 파일만 처리가 가능하도록 되어 있다.
- loader 를 이용하여 다른 형태의 웹 자원들을 (img, css, ...) js 로 변환하여 로딩

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

- loader에서 모듈 로딩 순서는 배열의 요소 오른쪽에서 왼쪽으로 진행된다.

```
{  
  test: /backbone/,  
  use: [  
    'expose-loader?Backbone',  
    'imports-loader?=underscore,jquery'  
    // 순서대로 (1) jquery , (2) underscore 로딩  
  ]  
}
```

- 위 설정 파일을 webpack으로 번들링 한 결과물은 아래와 같다.

```
var __WEBPACK_AMD_DEFINE_ARRAY__, __WEBPACK_AMD_DEFINE_RESULT__;//*** IMPORTS FROM import  
var _ = __webpack_require__(0);  
var jquery = __webpack_require__(1);
```

## Loader 더 많은 설정

## Babel Loader - ES6

- preset : Babel 플러그인 리스트

```
module: {
  rules: [{
    test: /\.js$/,
    use: [
      {
        loader: 'babel-loader',
        options: {
          presets: [
            ['es2015', 'react', {modules: false}] // Tree Shaking : 모듈에서 사용되지 않은 코드
          ]
        }
      }
    ]
  }]
}
```

```
//.bablerc
{
  "presets": ["react", "es2015"]
}
```

## 짚고 넘어가기 - Webpack 설정

- webpack 설정 값 여러개 받기

```
require("!style!css!./style.css"); // ! 로 여러 설정값을 받을 수 있다.  
require("./style.css");
```

# 실습 - Example 1

## Code Splitting 실습

### 실습 절차

1. `npm init -y` 으로 package.json 생성
2. npm 명령어로 loader & plugin 설치
3. `index.html` , `app/index.js` , `base.css` 생성
4. `webpack.config.js` 생성
5. `webpack` 실행

## ExtractTextWebpackPlugin 로 css 파일 분리 실습

- CSS 를 bundle.js 파일 안에 번들링 하지 않고, 빌드시에 별도의 .css 파일로 분리해준다.

```
var ExtractTextPlugin = require("extract-text-webpack-plugin");

module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ExtractTextPlugin.extract({
          fallback: "style-loader",
          use: "css-loader"
        })
      }
    ]
  },
  plugins: [
    new ExtractTextPlugin("styles.css"),
  ]
}
```

# Webpack Plugins

플러그인은 파일별 커스텀 기능을 사용하기 위해서 사용한다.

- ex) JS minification, file extraction, alias (별칭)

```
module.exports = {
  entry: {},
  output: {},
  module: {},
  plugins: [
    new webpack.optimize.UglifyJsPlugin()
    // ...
  ]
};
```

## ProvidePlugins

- 모든 모듈에서 사용할 수 있도록 해당 모듈을 변수로 변환한다.

```
new webpack.ProvidePlugin({  
  $: "jquery"  
})
```

## DefinePlugin

- Webpack 번들링을 시작하는 시점에 사용 가능한 상수들을 정의한다.
- 일반적으로 개발계 & 테스트계에 따라 다른 설정을 적용할 때 유용하다

```
new webpack.DefinePlugin({
  PRODUCTION: JSON.stringify(true),
  VERSION: JSON.stringify("5fa3b9"),
  BROWSER_SUPPORTS_HTML5: true,
  TWO: "1+1",
  "typeof window": JSON.stringify("object")
})
```

## ManifestPlugin

- 번들링시 생성되는 코드 (라이브러리)에 대한 정보를 json 파일로 저장하여 관리

```
new ManifestPlugin({  
  fileName: 'manifest.json',  
  basePath: './dist/'  
})
```

## 실습 - Example 2

### Libraries Code Splitting

#### 실습절차

1. `npm init -y` 으로 package.json 생성
2. npm 명령어로 loader & plugin 설치
3. `index.html` , `app/index.js` 생성

#### 4. webpack.config.js 생성

- main & vendor 분할에 따른 filename 지정

```
module.exports = {
  entry: {
    main: './app/index.js',
    vendor: [
      'moment',
      'lodash'
    ],
  },
  output: {
    filename: '[name].js',
    path: path.resolve(__dirname, 'dist')
  },
}
```

- CommonsChunkPlugin 을 사용한 vendor 라이브러리 추출

```
plugins: [
  new webpack.optimize.CommonsChunkPlugin({
    name: 'vendor' // Specify the common bundle's name.
  }),
]
```

- ManifestPlugin 으로 vendor 라이브러리와 실제 앱 로직 js 분리. json 파일로 번들링 파일 관리

```
new ManifestPlugin({
  fileName: 'manifest.json',
  basePath: './dist/'
})
```

## 5. webpack 실행

## 개발자 도구 연동

- 브라우저에서 webpack 으로 컴파일된 파일을 디버깅 하기란 어려움
- 따라서, 아래와 같이 source-map 설정을 추가하여 원래의 파일구조에서 디버깅이 가능

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

[devtool official doc](#)

## wepback 빌드를 위한 개발 서버 구성

- `webpack-dev-server` : webpack 자체에서 제공하는 개발 서버이고 빠른 리로딩 기능 제공
- `webpack-dev-middleware` : 서버가 이미 구성된 경우에는 webpack 을 미들웨어로 구성하여 서버와 연결

개인 프로젝트에는 시작하기 쉬운 `webpack-dev-server` 를 활용!

# Webpack Dev Server

- 페이지 자동고침을 제공하는 webpack 개발용 node.js 서버

## 설치 및 실행

- 아래 명령어로 dev-server 설치

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

- 설치 후 아래 명령어로 서버 실행

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

- 또는 package.json 에 아래와 같이 명령어를 등록하여 간편하게 실행 가능

```
"scripts": { "start": "webpack-dev-server" }
```

추가 옵션 설정은 [여기](#)를 참고

## Options

- `publicPath` : Webpack 으로 번들한 파일들이 위치하는 곳. default 값은 `/`

```
// 항상 `/` 를 앞뒤에 붙여야 한다.  
publicPath: "/assets/"
```

- `contentBase` : 서버가 로딩할 static 파일 경로를 지정. default 값은 `working directory`

```
// 절대 경로를 사용할 것  
contentBase: path.join(__dirname, "public")  
// 비활성화  
contentBase: false
```

- `compress` : gzip 압축 방식을 이용하여 웹 자원의 사이즈를 줄인다.

```
compress: true
```

# 실습 - Example 3

## Webpack Dev Server

### 실습절차

1. `npm init -y` 으로 `package.json` 생성
2. `scripts` 에 `start` 명령어 추가
3. `npm` 명령어로 loader & plugin 설치
4. `index.html` , `app/index.js` 생성
5. `webpack.config.js` 생성
6. `npm start` 실행

## Webpack Dev Middleware

- 기존에 구성한 서버에 webpack에서 컴파일한 파일을 전달하는 middleware wrapper
- webpack에 설정한 파일을 변경시, 파일에 직접 변경 내역을 저장하지 않고 메모리 공간을 활용한다.
- 따라서, 변경된 파일 내역을 파일 디렉토리 구조안에서는 확인이 불가능하다.

## 설치

- 아래 명령어로 설치

```
npm install --save-dev express webpack-dev-middleware
```

- 설치후 webpack & webpack dev middle ware 등 로딩

```
var express = require("express");
var webpack = require("webpack");
var webpackDevMiddleware = require("webpack-dev-middleware");
var webpackConfig = require("./webpack.config");
```

- webpackDevMiddleware 에 config 세팅 적용 및 번들링 파일 경로 지정

```
var app = express();
var compiler = webpack(webpackConfig);

app.use(webpackDevMiddleware(compiler, {
  publicPath: webpackConfig.output.publicPath // 일반적으로 output 에 설정한 publicPath 와 동
  stats: {colors: true} // 번들링 시 webpack 로그 컬러 하이라이팅
  // lazy: true, // entry point 에 네트워크 요청이 있을 때만 컴파일을 다시한다.
}));

app.listen(3000, function () {
  console.log("Listening on port 3000!");
});
```

## Public Path 되짚고 넘어가기

- 여태까지 **Public Path** 는 모두 로컬의 정적인 파일이나, 로컬 서버의 환경에서 접근한 사례
- 아래는 실제 앱을 배포하여 CDN 으로 접근할 때의 설정
- 플러그인을 이용하여 Production Build 시에 URL 업데이트를 도와줌.
  - ex) CSS 의 url 속성 `./test.png` 의 앞에 Public Path (URL) 삽입

main.css loads an image 'test.png'

```
1 .image {  
2   height:32px;  
3   width:32px;  
4   background-image: url('./test.png');  
5 }  
6 }
```

main.js imports main.css as a dependency

```
1 import React from 'react';  
2 import {render} from 'react-dom';  
3 import Greeter from './Greeter';  
4 //simply import CSS so the Webpack know that it's a dependency  
5 import './main.css';  
6  
7 render(<Greeter />, document.getElementById('root'));  
8
```

#### Webpack.config.js

- Has publicPath set
- Has url-loader, which is publicPath "aware"

```
module.exports = {  
  //devtool: 'eval-source-map',  
  
  entry: __dirname + "/app/main.js",  
  output: {  
    path: __dirname + "/public",  
    publicPath: 'http://localhost:5000/', //Simulate CDN  
    filename: "bundle.js"  
  },  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        loader: 'style!css'  
      },  
      {  
        test: /\.png$/,  
        loader: "url-loader?limit=1"  
      },  
      {  
        test: /\.jpg$/  
      }  
    ]  
  }  
};
```

Output: url-loader updates the test.png's url to use publicPath

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="utf-8">  
    <title>Webpack Sample Project</title>  
    <style type="text/css">  
      body{margin:0;background-color:pink}.image{height:32px;width:32px;background-image:url(http://localhost:5000/9c938a4dc2701d09908707aebc48cf1a.png)}  
    </style>  
  </head>  
  <body></body>  
</html>
```

```
// Development: Both Server and the image are on localhost
.image {
  background-image: url('./test.png');
}
// Production: Server is on Heroku but the image is on a CDN
.image {
  background-image: url('https://someCDN/test.png');
}
```

# 실습 - Example 4

## Webpack Dev Middleware

### 실습절차

1. `npm init -y` 으로 `package.json` 생성
2. npm 명령어로 plugin 설치
3. `server.js` 생성 및 express & ejs 추가
4. `index.html` , `app/index.js` 생성
5. `webpack.config.js` 생성
6. `server.js` 실행

## Webpack Resolve

- Webpack 의 **모듈 번들링** 관점에서 봤을 때, 모듈 간의 의존성을 고려하여 모듈을 로딩해야 한다.
- 따라서, 모듈을 어떤 위치에서 어떻게 로딩할지에 관해 정의를 하는 것이 바로 Module Resolution

```
// 일반적인 모듈 로딩 방식
import foo from 'path/to/module'
// 또는
require('path/to/module');
```

그렇다면 여기서 우리가 주목해야 할 부분은 바로 "모듈을 어떻게 로딩해오느냐?"라는 점

## 1. 절대경로를 이용한 파일 로딩

- 파일의 경로를 모두 입력해준다.

```
import "/home/me/file";
import "C:\\Users\\me\\file";
```

## 2. 상대경로를 이용한 파일 로딩

- 해당 모듈이 로딩되는 시점의 위치에 기반하여, 상대 경로를 절대 경로로 인식하여 로딩한다.

```
import "../src/file1";
import "./file2";
```

# Resolve Option

config 파일에 `resolve` 를 추가하여 모듈 로딩에 관련된 옵션 사용

## alias

- 특정 모듈을 로딩할 때 `alias` 옵션을 이용하면 별칭으로 더 쉽게 로딩이 가능하다.

```
alias: {  
  Utilities: path.resolve(__dirname, 'src/path/utilities/')  
}  
  
import Utility from '../../src/path/utilities/utility';  
// alias 사용시 '/src/path/utilities/' 대신 'Utilities' 활용  
import Utility from 'Utilities/utility';
```

## modules

- `require()` `import ''` 등의 모듈 로딩시에 어느 폴더를 기준할 것인지 정하는 옵션

```
modules: ["node_modules"] // defaults
modules: [path.resolve(__dirname, "src"), "node_modules"] // src/node_modules 위치
```

# 실습 - Example 5

## Webpack Plugins & Resolve

### 실습절차

1. `npm init -y` 으로 package.json 생성
2. npm 명령어로 plugin 설치
3. `index.html` , `app/index.js` 생성
4. `webpack.config.js` 생성
5. `webpack` 실행
6. `app/index.js` , `webpack.config.js` 변경하여 alias & Provide 동작 확인

## Webpack watch 옵션

webpack 설정에 해당되는 파일의 변경이 일어나면 자동으로 번들링을 다시 진행

```
webpack --progress --watch
```

참고 : `npm install --save-dev serve` 한 후 아래처럼 `package.json` 에 명령어 설정 가능

```
"scripts": {  
  "start": "serve"  
}
```

## Gulp 연동

- Gulp 와 Webpack 모두 Node.js 기반이기 때문에 통합해서 사용하기 쉽다.

```
var gulp = require('gulp');
var webpack = require('webpack-stream');
var webpackConfig = require('./webpack.config.js');

gulp.task('default', function() {
  return gulp.src('src/entry.js')
    .pipe(webpack(webpackConfig))
    .pipe(gulp.dest('dist/'));
});
```

## Hot Module Replacement

- 웹 앱에서 사용하는 JS 모듈들을 갱신할 때, 화면의 새로고침 없이 뒷 단에서 변경 및 삭제 기능을 지원
- 공식 가이드에는 React 를 기준으로 사용법이 작성되어 있으므로 [참고](#)

## 참고 API

### path.join()

- 해당 API 가 동작되는 OS 의 파일 구분자를 이용하여 파일 위치를 조합한다.

```
path.join('/foo', 'bar', 'baz/asdf');
// Returns: '/foo/bar/baz/asdf'
```

## path.resolve()

- join()의 경우 그냥 문자열을 합치지만, resolve은 오른쪽에서 왼쪽으로 파일 위치를 구성해가며 유효한 위치를 찾는다.
- 만약 결과 값이 유효하지 않으면 현재 디렉토리가 사용된다. 반환되는 위치 값은 항상 absolute URL이다.

```
path.resolve('/foo/bar', './baz');
// Returns: '/foo/bar/baz'
```

```
path.resolve('/foo/bar', '/tmp/file/');
// Returns: '/tmp/file'
```

```
path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif');
// if the current working directory is /home/myself/node,
// this returns '/home/myself/node/wwwroot/static_files/gif/image.gif'
```

# Reference

- [Webpack2 Doc](#)
- [Webpack1 Doc](#)
- [webpack-howto](#)
- [webpack-howto2](#)
- [webpack beginners guide, Site Point](#)
- [requireJS-to-webpackConfig](#)
- [migration from requirejs to webpack](#)
- [webpack-shimming](#)
- [Critical-Dependencies](#)
- [Gulp Webpack plugin](#)

- [from require to webpac](#)
- [Webpack Dev Server StackOverFlow](#)
- [Webpack Dev Middleware in Express](#)
- [Webpack Confusing Part](#)
- [Regular Expression, MDN](#)
- [Regular Expression Test](#)
- [Webpack First Principle, Video](#)

끝