

# 성능 최적화 팁과 요령

# 개요

- JS 코딩 시 쉽게 적용할 수 있는 코딩 기법을 학습
- Chrome 브라우저 엔진 관점의 JS 코딩 기법을 학습
- 웹 자원 압축의 필요성을 이해하고, 압축 방법을 학습
- 웹 자원 캐싱을 위해 필요한 지식과 설정 방법을 학습

# 목차

- Javascript Best Practices
  - 실습
- Chrome Engines JS Tuning
- Web Resources Compression
  - HTML & JS Compression
  - Image Compression
  - Gzip Compression (실습)
- Caching
  - Deeper in HTTP Header & Request

# Javascript Best Practices

## 단순한 if else 구문 대신에 삼항연산자 사용

```
var mad = true;
var hulk;

if(mad) {
  hulk = "incredible";
} else {
  hulk = "doctor";
}

// 삼항연산자 이용
var hulk = mad ? "incredible" : "doctor";
// 변수 두개 이상시, 즉시 실행함수
mad && crazy ? function () {
  // ...
}();
:function () {
  // ...
}();
```

## 논리 연산자 or 과 and 의 특징

코드 결과 계산 시 or는 좌측 값 우선, and는 우측 값 우선

```
function addElements(elements) {  
    // 일반적인 유효 값 검사  
    if (this.elements === null) {  
        this.elements = [];  
    }  
  
    // OR 연산자를 활용한 값 검사  
    this.elements = this.elements || [];  
}
```

```
var a = 10;  
var b = 20;  
var result1 = a && b;  
console.log(result1); // 20  
var result2 = b && a;  
console.log(result2); // 10
```

## if - else 와 switch 구문

if else문은 순서대로 실행. switch는 해당 조건으로 바로 분기

```
var a = 3;  
if (a === 1) { }  
else if (a === 2) { }  
else if (a === 3) { }  
  
switch (a) {  
  case 1:  
    ...  
  case 2:  
    ...  
  case 3:  
    ...  
}
```

## 반복문 최적화

```
var puzzles = {  
    pieces: ["A", "B", "C", "D"]  
};  
  
for (var i = 0; i < puzzles.pieces.length; i++) {  
    console.log(puzzles.pieces[i]);  
}
```

위 반복문에서 메모리 연산이 수행되는 부분은 다음과 같다.

1. i 값 검색
2. puzzles 객체 탐색
3. pieces 속성 탐색
4. pieces 배열 인덱스 검색
5. length 프로퍼티 검색

## 이전 반복문을 최적화 하면

```
// 1 - x 로 불필요한 반복 접근수 감소
var x = puzzles.pieces.length;
for (var i = 0; i < x; i++) {
  console.log(puzzles.pieces[i]);
}

// 2 - 전역 변수 메모리 절약
for (var i = 0, x = puzzles.pieces.length; i < x; i++) {
  console.log(puzzles.pieces[i]);
}

// 3 - 객체 접근 연산 수 감소
var list = puzzles.pieces;
for (var i = 0, x = puzzles.pieces.length; i < x; i++) {
  console.log(list[i]);
}
```

## 함수의 메서드 재활용 (상속 관점)

```
function calculator(number) {  
    this.number = number;  
    this.add = function () {},  
    this.sub = function () {}  
}
```

```
calculator.prototype = {  
    add: function () {},  
    sub: function () {}  
}
```

## Prototype 확인 코드

# 실습 - JS Best Practices

위에서 배운 삼항연산자, 논리연산자, 반복문, 프로토타입 을 아래 코드에 직접 적용합니다.

## 실습 절차

1. [DevCampWAPQuiz](#) Github Repository Fork
2. Fork 한 repository Clone 하여 로컬에 저장
3. js-best-practices 아래에 자기 이름으로 폴더 생성 ex) jangkeehyo
4. 아래 파일들을 이용하여 답안 작성 후 Pull Request 요청
  - practice.js
  - prototype.js

# Chrome V8 Engines 관점의 JS 코딩 기법

## Hidden Class 사용

- V8은 런타임시에 객체 처리를 위해 내부적으로 Hidden class를 만들어서 사용한다.
- Javascript는 런타임시에 데이터 타입을 변경할 수 있다.

```
// 성능 향상을 위해 엔진에서 생성하는 Hidden Class 이외에 별도의 Hidden Class 를 생성
function Point(x, y) {
    this.x = x;
    this.y = y;
}

var p1 = new Point(11, 22);
var p2 = new Point(33, 44);
// 위 p1, p2 는 위에 선언한 Hidden Class 를 공용으로 사용.
p2.z = 55;
// p2 가 바라보는 Hidden Class 가 바뀌어 별도의 Hidden Class 로 생성하여 처리해야 함
```

## Array Usage

Javascript 일반 배열의 경우 초기 값을 주는 것이 더 효과적

- 키 값이 순서대로 정해졌을 때 사용되는 선형 저장소
- 순서가 정해지지 않았을 때 사용하는 해쉬 저장소

```
// 해쉬 타입 배열 저장소
```

```
a = new Array();
for (var b = 0; b < 10; b++) {
    a[0] = b;
}
```

```
// 선형 타입 배열 저장소
```

```
a = new Array();
a[0] = 0;
for (var b = 0; b < 10; b++) {
    a[0] = b; // 배 이상의 속도가 차이남
}
```

## 불필요한 배열 타입 변환 및 할당을 피하기 위해 배열 선언시 리터럴로 선언

```
var arr = new Array();
arr[0] = 0; // 일반 배열
arr[1] = 1.1; // 일반 -> Double 타입 배열
arr[2] = 2; // Double 타입 배열 -> 일반

// Javascript 는 런타임시에 타입이 변경되는 속성이 있다.
// 따라서, 배열 리터럴을 이용하여 아래와 같이 작성
var arr = [0, 1.1, 2];
```

## V8 엔진의 컴파일러

- 초기 자바스크립트 언어는 인터프리터 형태, 최근에는 JS 런타임 엔진이 컴파일 방식을 사용

### 자바스크립트 JIT(Just in Time) 컴파일러의 2 가지 종류

- *Full Compiler* : 일반적인 자바스크립트 코드로 변환
- Optimizing Compiler : 양질의 자바스크립트 코드로 변환 (더 긴 컴파일 시간)

## Full Compiler

- 모든 코드에서 동작하고 빠르게 코드를 실행시키지만 코드 품질이 우수하진 않음
- 컴파일 시점에 데이터 타입에 대한 가정을 하지 않음 - 변수의 데이터 타입이 런타임시에 변경된다고 간주
- 연산시에 형변환이 자주 일어나지 않도록 주의하며 구현

```
// 내부적으로 생성한 Hidden Class
function add(x, y) {
  return x + y;
}

add(1, 2);      // 위 함수의 타입을 숫자로 지정
add("a", "b"); // 숫자로 지정된 위 함수를 사용하지 않음
```

# Web Resources Compression

## Image 압축

- 불필요한 이미지 리소스 제거
- 대체 가능한 경우에는 CSS3 사용. ex) 애니메이션 등
- 이미지 안에 들어가는 텍스트 대신 웹 폰트 사용

효율적인 이미지 활용에 대해서 늘 재고하는 자세가 필요

## Vector vs Raster

- Vector 이미지 : 선, 점 폴리곤을 이용하여 나타냄
  - ex) SVG (로고, 텍스트, 아이콘)
- Raster 이미지 : 픽셀의 값을 인코딩하여 이미지를 나타냄
  - ex) PNG, JPEG, GIF (쇼핑몰 상품이나 일반 사진)

SVG 최적화는 [svgo](#) 활용

[WebP](#) 이미지 참고

## Image 활용

형식	투명도	애니메이션	브라우저
GIF	지원	지원	전체
PNG	지원	지원 안 함	전체
JPEG	지원 안 함	지원 안 함	전체
JPEG XR	지원	지원	IE
WebP	지원	지원	Chrome, Opera, Android

- GIF : 애니메이션 활용에 사용
- JPEG : 압축으로 사이즈를 크게 줄일 수 있음. 품질과 사이의 균형유지가 필요
- PNG : 고품질의 사진. 압축 불가능. 상대적으로 큰 사이즈

이미지 안에 들어간 텍스트는 변경이 안되므로 주의!

## Image 크기

- 이미지 파일의 실제 크기보다 더 작게 웹페이지에 로딩되는 경우 불필요한 오버헤드가 발생

화면 해상도	실제 크기	표시 크기(CSS px)	불필요한 픽셀
1x	110 x 110	100 x 100	$110 \times 110 - 100 \times 100 = 2100$
1x	410 x 410	400 x 400	$410 \times 410 - 400 \times 400 = 8100$
1x	810 x 810	800 x 800	$810 \times 810 - 800 \times 800 = 16100$
2x	220 x 220	100 x 100	$210 \times 210 - (2 \times 100) \times (2 \times 100) = 8400$
2x	820 x 820	400 x 400	$820 \times 820 - (2 \times 400) \times (2 \times 400) = 32400$
2x	1620 x 1620	800 x 800	$1620 \times 1620 - (2 \times 800) \times (2 \times 800) = 64400$

따라서, 웹 페이지에서 보여지는 이미지 크기와 실제 이미지 크기가 최대한 같아야 한다.

## Image 최적화 요약

- 확대 / 축소에 강한 SVG 활용
- 사이트의 특성과 요구사항에 맞게 압축을 활용
- 실제 이미지 파일 크기와 웹 페이지의 이미지 크기를 동일하게...
- Gulp, Grunt 와 같은 자동화 도구를 이용하여 항상 이미지는 압축!

# HTML & JS Compression

- CRP : 빠른 웹 페이지 로딩을 위해서는 CRP Length & Critical Byte 를 최소화
- 서버에 요청하는 HTTP Request 수 와 서버로부터 다운 받을 데이터의 크기가 작을수록 로딩속도는 빨라진다.

일반적인 텍스트 데이터의 압축 방식 3가지

1. Minification : 공백 및 주석 제거, Byte 감소
2. Concatenation : 여러 개의 js 파일을 하나로 병합, Request 감소
3. Compression : 공백 및 주석 제거 & 변수명 짧게 변경, Byte 감소

위 관련 실습은 성능 최적화 도구 시간에 진행

## Gzip 압축

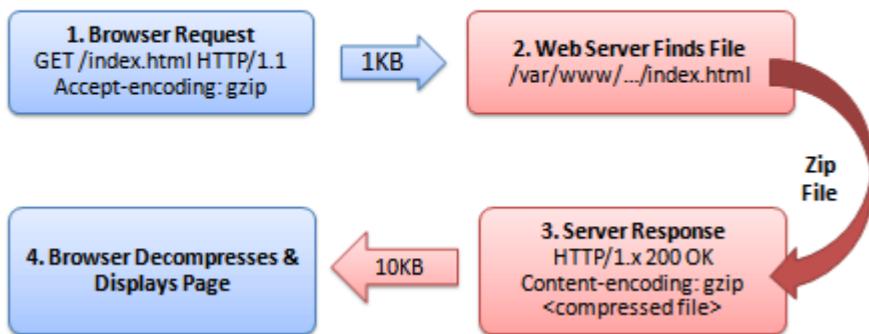
- 일반적인 웹 자원 압축은 빌드 자동화 도구를 활용한 리소스 파일 크기 자체의 압축
- Gzip 압축은 한정된 네트워크 대역폭과 지연시간을 효율적으로 활용하기 위한 측면
- Client (브라우저) - Server 간 데이터 전송량을 줄일 수 있다.

| 참고 : [브라우저별 지원 여부](#)

## Gzip 의 동작 원리

1. Gzip 을 지원하는 브라우저에서 자원 요청
2. Gzip 설정이 된 서버에서 응답을 받아 해당 자원을 압축 후 반환
3. 브라우저에서 파일 수신 및 압축 해제 후 표시

### Compressed HTTP Response



## Node.js 의 Gzip 압축

- [Node.js](#) : Javascript 로 구현 가능한 Server Side 프레임워크. 이벤트 드리븐 방식
- Node.js 의 설정에 아래의 패키지를 설치하여 추가해준다.
  - Node v2.x : [gippo](#)
  - Node v3.x : [compression](#)

## Node.js 를 이용한 Gzip 압축 실습

### 실습절차

1. [PAO Repository clone](#)
2. gzip-nodejs 폴더 이동 후 `npm install` 입력
3. 설정된 node module 모두 설치 후 `node server.js` 로 서버 실행

## Tomcat 의 Gzip 압축 설정

```
<Connector port="8080" protocol="HTTP/1.1" redirectPort="8443"
    URIEncoding="UTF-8"
    connectionTimeout="20000"
    compression="on"
    compressionMinSize="1024"
    noCompressionUserAgents="ozilla, traviata"
    compressableMimeType="text/html,text/xml,text/plain,text/javascript,text/css,application/javascript,application/x-javascript"/>
```

- `compression` : 압축 사용 여부. "off" (default)
- `compressionMinSize` : 압축하는 최소 파일 크기. "2048" (default)
- `noCompressionUserAgents` : 압축을 사용하지 않을 브라우저 지정. "" (default)
- `compressableMimeType` : 압축을 사용 할 파일 타입. "" (default)

## HTTP Request & Response Header

- HTTP Header 속성 (Encoding-Header, Cache, Etag, Connection, ... )
- [HTTP 상태 코드](#) summary

---

▼ Response Headers [view source](#)

```
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Connection: keep-alive
Date: Fri, 19 May 2017 04:14:10 GMT
ETag: W/"17-15c1dde4669"
Last-Modified: Thu, 18 May 2017 23:22:05 GMT
X-Powered-By: Express
```

---

## HTTP Header Encoding

크롬 개발자도구 Network 패널의 파일을 클릭하면 아래와 같이 표시

Accept-Encoding : Client 에서 Server 로 보내는 요청. 헤더에 명시된 인코딩 값 (압축) 을 이해하고, 디코딩 (압축 해제) 를 수행할 수 있다는 것을 서버에 알림

- ex) gzip, deflate : gzip, deflate 압축 방식을 수용할 수 있으므로, Server 에서 압축해서 보낸 파일을 해제하여 브라우저에 표시할 수 있다.

Content-Encoding : Server 에서 보내는 응답이 어떤 인코딩 방식, 즉 어떤 방식으로 압축되었는지 표시.

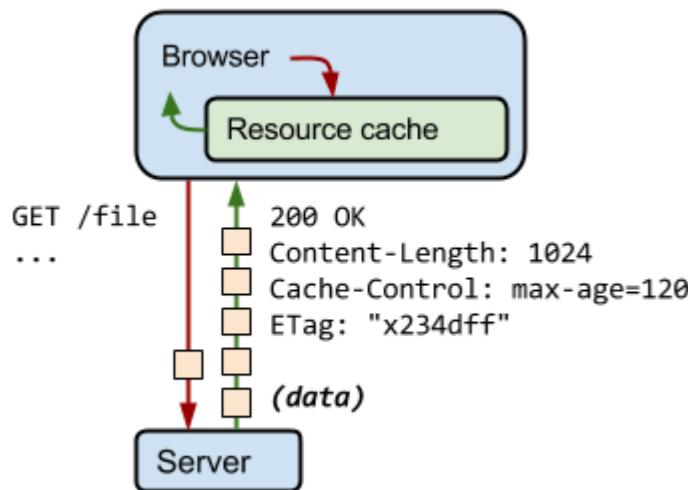
- ex) gzip : 해당 응답은 Gzip 으로 압축하였으니, Client (브라우저) 에서 해제해서 표시하길 바람

## HTTP Connection

- 지속 연결 : `Connection : Keep-Alive` , 하나의 TCP 커넥션을 열고 모든 요청을 처리 (HTTP 1.1)
- 비지속 연결 : `Connection : close` , 매번 자원 요청시 새로운 TCP 커넥션 생성 (HTTP 1.0)

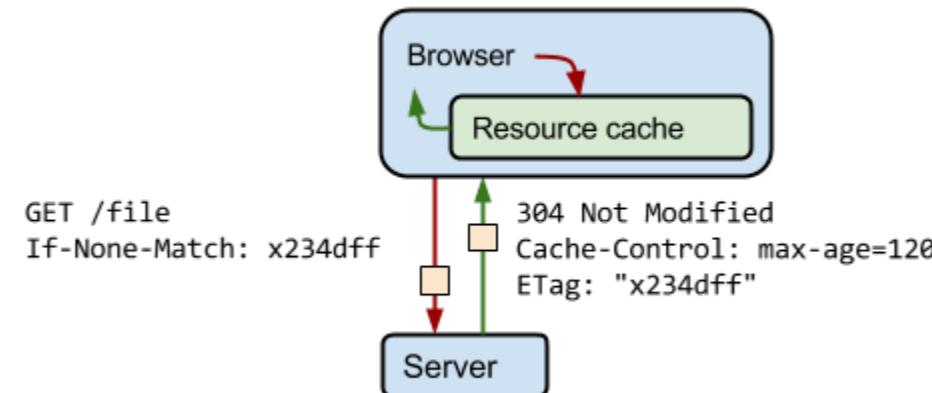
# Caching

- 불필요한 Client - Server 간의 네트워크 왕복 요청을 줄이기 위해 필요한 기법
- 일반적으로 최신 브라우저에 모두 HTTP 캐싱이 다 구현되어 있음
- 개발자는 HTTP Header 에 적절한 캐싱 설정값을 세팅해주기만 하면 된다.



## E-tag

- 리소스 유효성 검사 태그로 해당 리소스의 갱신 필요여부를 확인하는 지문 역할 (최신 버전의 자원인가?)
- 웹 페이지의 리소스가 변경되지 않음을 Client - Server 간의 특정 값으로 검사
- 동작 방식
  - i. Server에서 해당 파일의 특정 해쉬값 발급
  - ii. Client에서 이 해쉬값을 받아 확인 후 파일이 변하지 않으면 그대로 다시 Server에 전송
  - iii. Server에서 받은 해쉬값이 변경되지 않으면 `304 Not Modified` 반환
  - iv. 해당 파일을 다시 전송해서 받는 과정이 생략되어 시간과 대역폭 절약



## Cache-Control

- 캐싱에 대한 옵션을 설정하는 속성
- `no-cache` : 해당 리소스에 대한 확인 요청을 보내고 캐쉬를 강제
- `no-store` : 해당 리소스는 절대 캐쉬하지 않음
- `public` : 사이트 로고 와 같은 다수의 인원이 필요한 정보에 대해 캐싱 설정
- `private` : 로그인 정보와 같이 캐쉬가 필요 없는 개인정보를 다룰 때 설정. 서버와의 통신시 대역폭을 줄이기 위한 전략
- `max-age` : 서버로부터 받은 데이터의 유효시간 길이 설정 (초 단위)

### Cache-Control 지시문 & 설명

<code>max-age=86400</code>	응답을 최대 1일(60초 x 60분 x 24시간) 동안 브라우저 및 중간 캐시(즉, 'public'임)가 캐시할 수 있음.
<code>private, max-age=600</code>	응답을 최대 10분(60초 x 10분) 동안 클라이언트의 브라우저만 캐시할 수 있음.
<code>no-store</code>	응답을 캐시할 수 없으며 매 요청마다 전체를 모두 가져와야 함

## 참고

- [Web Fundamentals - Google](#)
- [HTML5 Rocks - Google](#)
- [HTTP Spec - W3C](#)
- [Gzip is not enough - Youtube](#)
- [Gzip compression](#)
- [How to optimize your site with Gzip compression](#)
- [Web Perf Engineering - SK Planet](#)

끝