# 13. 캐싱 및 성능 개선

## Spring Cache (`@Cacheable`, `@CacheEvict`)

Spring Cache는 애플리케이션 성능 최적화를 위해 자주 사용되는 데이터의 조회 결과를 **캐시**에 저장하고 재사용할 수 있게 해주는 추상화 기능이다.
이를 통해 DB나 API 호출 등 상대적으로 비용이 큰 연산의 횟수를 줄여서 **응답 속도를 개선**하고 **시스템 부하를 감소**시킬 수 있다.

Spring Boot는 기본적으로 Spring Cache abstraction(`org.springframework.cache`)을 제공하며, 다양한 Cache Provider(예: Redis, EhCache, Caffeine 등)와 쉽게 연동할 수 있도록 구성되어 있다.

Spring Cache의 주요 어노테이션으로는 `@Cacheable`, `@CacheEvict`, `@CachePut` 등이 있다.

---

### @Cacheable

`@Cacheable`은 메서드 실행 결과를 **캐시에 저장**하고,
다음 번 동일한 인자에 대한 호출 시 **캐시된 값을 반환**하도록 한다.
즉, **읽기(Read)** 시 캐시 사용에 해당한다.

```
@Cacheable(value = "books", key = "#isbn")
public Book findBook(String isbn) {
    simulateSlowService();
    return bookRepository.findByIsbn(isbn);
}
```

- `value` → 캐시 이름 (Cache Manager가 이 이름으로 Cache를 구분)
- `key` → 캐시의 키 (동일 key가 있을 경우 캐시에서 바로 반환)
- 첫 호출 → `bookRepository.findByIsbn()` 실행 결과를 캐시에 저장
- 두 번째 호출 → 동일 isbn이면 캐시에서 결과 반환 (메서드는 실행 안됨)

---

### @CacheEvict

`@CacheEvict`는 **캐시에서 값을 제거**하는 데 사용된다.
주로 **데이터가 변경될 때**(수정/삭제) 캐시를 무효화하여
잘못된 데이터를 반환하지 않도록 하기 위해 사용한다.

```
@CacheEvict(value = "books", key = "#isbn")
public void deleteBook(String isbn) {
    bookRepository.deleteByIsbn(isbn);
}
```

- `deleteBook()` 호출 시 해당 isbn에 대한 캐시도 함께 제거된다.
- `allEntries = true`를 지정하면 **캐시 전체 삭제**도 가능하다.

```java
@CacheEvict(value = "books", allEntries = true)
public void clearCache() {
    // 모든 books 캐시를 삭제
}
```

---

## @CachePut

`@CachePut`은 메서드는 **항상 실행되지만 결과는 캐시에 저장**한다.
`@Cacheable`은 조건이 충족되면 메서드 실행을 건너뛰지만,
`@CachePut`은 항상 실행된다.

주로 **업데이트 후 새로운 값을 캐시에 반영**할 때 사용한다.

```java
@CachePut(value = "books", key = "#book.isbn")
public Book updateBook(Book book) {
    return bookRepository.save(book);
}
```

---

## CacheManager와 CacheProvider

Spring Boot에서는 다양한 CacheProvider와 연동할 수 있다:

- SimpleCacheManager (기본, ConcurrentHashMap 기반 메모리 캐시)
- EhCache / EhCache 3
- Caffeine
- RedisCacheManager (Redis 연동 시 주로 사용)
- JCache (JSR-107)

### 기본 설정 (application.yml)

```yaml
spring:
  cache:
    type: redis
```

혹은

```yaml
spring:
  cache:
    type: caffeine
```

---

## Cache 설정 시 주의점

- 캐시 대상은 반드시 **정적이거나 자주 변경되지 않는 데이터**에 적용한다.
- 캐시 무효화 전략이 중요하다. 데이터 변경 시 적절히 `@CacheEvict` 또는 `@CachePut`을 사용하여 관리해야 한다.
- 메모리 캐시 사용 시 OOM(OutOfMemory)을 방지하기 위한 TTL(Time To Live) 설정이 필요하다.
- Redis 등 분산 캐시 사용 시 **네트워크 비용과 일관성 이슈**도 고려해야 한다.

---

## 간단한 흐름도

```
요청 발생 → 캐시 확인
    └─> 있음 → 캐시된 결과 반환
    └─> 없음 → 실제 메서드 실행 → 결과 캐시 저장 → 반환
```

---

## 전체 예제

```
@Service
public class BookService {

    @Cacheable(value = "books", key = "#isbn")
    public Book findBook(String isbn) {
        simulateSlowService();
        return bookRepository.findByIsbn(isbn);
    }

    @CacheEvict(value = "books", key = "#isbn")
    public void deleteBook(String isbn) {
        bookRepository.deleteByIsbn(isbn);
    }

    @CachePut(value = "books", key = "#book.isbn")
    public Book updateBook(Book book) {
        return bookRepository.save(book);
    }
}
```

---

이렇게 `@Cacheable`, `@CacheEvict`, `@CachePut`을 잘 조합하면
복잡한 로직 없이 빠르게 캐시 기능을 적용할 수 있다.
단, **캐시의 일관성 관리 전략**은 반드시 명확하게 수립해야 한다.

# Redis 연동

Spring Boot는 **Spring Cache abstraction**을 통해 다양한 Cache Provider와 손쉽게 통합할 수 있다. 그중에서도 Redis는 **고성능 인메모리 데이터 저장소**로서, 분산 캐시 구축과 확장성 확보 측면에서 매우 유용하다.
본 절에서는 Spring Boot 애플리케이션에 Redis Cache를 적용하는 기본적인 흐름과 구성 방법을 설명한다.

---

## 1️⃣ 의존성 추가

Spring Boot에서는 `spring-boot-starter-data-redis` 스타터를 사용하여 Redis와의 통합을 지원한다. 기본적으로 **Lettuce 클라이언트**가 포함되어 있으며, 필요에 따라 Jedis로 교체할 수 있다.

### Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

### Gradle

```
implementation 'org.springframework.boot:spring-boot-starter-data-redis'
```

---

## 2️⃣ Redis 서버 준비

Redis 서버는 로컬 환경에 설치하거나 Docker 기반으로 쉽게 실행할 수 있다.

### Docker 예시

```
docker run -p 6379:6379 redis
```

기본 포트는 `6379`이며, 초기 설정 시 인증 없이 접속 가능하다.

---

## 3️⃣ Spring Boot 설정

Redis 연동을 위한 기본 설정은 `application.yml` 또는 `application.properties` 파일에 작성한다.

```
spring:
  cache:
    type: redis  # Spring Cache가 Redis를 CacheManager로 사용
  redis:
    host: localhost
    port: 6379
    # password: your_redis_password (필요 시 사용)
```

`spring.cache.type=redis` 설정이 핵심이다. 이를 통해 **Spring Cache abstraction**은 내부적으로 RedisCacheManager를 활용하게 된다. 🎯

---

## 4️⃣ Cache 어노테이션 활용

Spring Cache 어노테이션은 CacheManager 설정에 따라 자동으로 Redis와 연계된다.

### @Cacheable 예시

```
@Cacheable(value = "books", key = "#isbn")
public Book findBook(String isbn) {
    return bookRepository.findByIsbn(isbn);
}
```

📌 지정된 `value`는 Redis 내 cache name이며, key는 해당 데이터의 고유 키가 된다.

### @CacheEvict 예시

```
@CacheEvict(value = "books", key = "#isbn")
public void deleteBook(String isbn) {
    bookRepository.deleteByIsbn(isbn);
}
```

📌 데이터 삭제 시 캐시 무효화도 동시에 이루어진다.

### @CachePut 예시

```
@CachePut(value = "books", key = "#book.isbn")
public Book updateBook(Book book) {
    return bookRepository.save(book);
}
```

📌 업데이트 후 Redis에 최신 데이터를 반영한다.

---

## 5️⃣ RedisCacheManager 커스터마이징

생성되는 Cache 항목의 만료 시간(TTL)을 설정하여 **불필요한 메모리 사용**과 **데이터 일관성 문제**를 방지할 수 있다. 이를 위해 CacheManager를 커스터마이징한다. 🛠️

```
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public RedisCacheConfiguration cacheConfiguration() {
        return RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10))  // TTL 10분 설정
            .disableCachingNullValues();
    }

    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory connectionFactory) {
        return RedisCacheManager.builder(connectionFactory)
```

```
            .cacheDefaults(cacheConfiguration())
            .build();
    }
}
```

### 주요 구성 요소

- `.entryTtl(Duration)`
  개별 캐시 항목의 생명 주기 설정.
- `.disableCachingNullValues()`
  `null` 값은 캐시 대상에서 제외.

이러한 구성을 통해 **정확하고 효율적인 캐시 전략**을 설계할 수 있다. ✅

---

## 6️⃣ RedisTemplate vs CacheManager

| 목적 | 사용 API | 특징 |
| --- | --- | --- |
| Spring Cache abstraction | `@Cacheable`, `@CacheEvict`, `@CachePut` | 고수준 추상화, 통합 관리 |
| Redis 직접 제어 | `RedisTemplate` | Key-Value, List, Hash 등 저수준 제어 가능 |

### RedisTemplate 사용 예시

```
@Autowired
private RedisTemplate<String, Object> redisTemplate;

public void saveValue(String key, Object value) {
    redisTemplate.opsForValue().set(key, value);
}

public Object getValue(String key) {
    return redisTemplate.opsForValue().get(key);
}
```

RedisTemplate은 **캐시 외 일반 데이터 처리**에 유용하다.
예를 들어 Session 관리, 분산락, 카운터 등 다양한 시나리오에 사용할 수 있다. 🗃️

---

## 7️⃣ 구성 흐름 요약

```
Controller → Service → @Cacheable 메서드 호출
   ↓
CacheManager 확인 → Redis에서 캐시 확인
   ↓
(있음) → Redis에서 값 반환
(없음) → 메서드 실행 → 결과 Redis에 저장 후 반환
```

## 8️⃣ 주의 사항 ⚠️

- Redis는 메모리 기반이므로 **용량 관리**가 매우 중요하다.
- TTL 전략을 반드시 구성할 것 (영구 저장 시 OOM 위험).
- Key 네이밍은 명확한 prefix 적용 권장 (`appname::cacheName::key` 형태).
- Redis Cluster 구성 시 **Failover** 및 **분산 전략** 고려 필요.

---

### 정리하자면:

✅ `spring-boot-starter-data-redis` 추가
✅ `application.yml` 구성
✅ `@Cacheable`, `@CacheEvict`, `@CachePut` 활용
✅ RedisCacheManager 커스터마이징으로 TTL 적용
✅ 필요 시 RedisTemplate 병행 사용

# EhCache, Caffeine 등 로컬 캐시

Spring Boot에서 제공하는 Spring Cache abstraction은 다양한 Cache Provider와 손쉽게 통합할 수 있다.
 그 중 **EhCache**와 **Caffeine**은 JVM 내에서 동작하는 **로컬(in-memory) 캐시**로서, 네트워크 지연 없이 매우 빠른 응답을 제공한다.

로컬 캐시는 다음과 같은 시나리오에 적합하다:

- 캐시 데이터가 노드마다 독립적으로 유지되어도 되는 경우
- 데이터 업데이트 빈도가 낮고 읽기 요청이 많은 경우
- 네트워크 I/O 부담을 줄이고자 하는 경우

---

## 1️⃣ EhCache 구성

### 1. 의존성 추가

EhCache 3.x는 JSR-107(JCache) 표준을 지원한다.

```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
</dependency>
```

### 2. application.yml 설정

```
spring:
  cache:
    type: jcache
```

### 3. ehcache.xml 구성

`src/main/resources/ehcache.xml` 파일을 작성한다.

```
<config xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xmlns='http://www.ehcache.org/v3'
        xsi:schemaLocation="http://www.ehcache.org/v3
http://www.ehcache.org/schema/ehcache-core.xsd">

    <cache alias="books">
        <expiry>
            <ttl unit="minutes">10</ttl>
        </expiry>
        <heap>1000</heap>
    </cache>

</config>
```

- `ttl` → 항목당 Time To Live 설정
- `heap` → JVM 내에서 유지할 최대 항목 수

### 4. 사용 예시

```
@Cacheable(value = "books", key = "#isbn")
public Book findBook(String isbn) {
    return bookRepository.findByIsbn(isbn);
}
```

📌 EhCache는 **네트워크 없이 JVM 내부**에서 빠른 속도로 캐시 제공.

---

## 2️⃣ Caffeine 구성 🚀

Caffeine은 최신 Java 기반으로 개발된 매우 빠른 로컬 캐시 라이브러리이다.
Google Guava Cache의 후계로서 성능과 유연성이 뛰어나다.

### 1. 의존성 추가

```
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
</dependency>
```

Spring Boot에서는 `spring-boot-starter-cache`에 포함된 CacheManager가 Caffeine과 통합 가능하다.

### 2. application.yml 설정

```yaml
spring:
  cache:
    type: caffeine
    caffeine:
      spec: maximumSize=1000,expireAfterWrite=10m
```

`spec` 속성에서 **최대 사이즈**와 **만료 정책** 등을 설정한다.

### 3. 사용 예시

```java
@Cacheable(value = "books", key = "#isbn")
public Book findBook(String isbn) {
    return bookRepository.findByIsbn(isbn);
}
```

### 4. 고급 설정 (Java Config 사용 시)

```java
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CaffeineCacheManager cacheManager() {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager("books");
        cacheManager.setCaffeine(Caffeine.newBuilder()
            .maximumSize(1000)
            .expireAfterWrite(10, TimeUnit.MINUTES));
        return cacheManager;
    }
}
```

---

## 3️⃣ EhCache vs Caffeine 비교 🧐

| 항목 | EhCache | Caffeine |
|---|---|---|
| 라이선스 | Apache 2.0 | Apache 2.0 |
| 주요 언어 | Java | Java |
| JSR-107 지원 | O (EhCache 3.x) | O |
| 속도 | 매우 빠름 | **더 빠름** (최적화된 알고리즘 사용) |
| 사용 용도 | 전통적 엔터프라이즈 시스템 | 최신 고성능 서비스 |
| 구성 방법 | xml 기반 구성 많이 사용 | Java 코드 기반 구성에 최적화 |
| 특징 | 디스크 저장 지원 (off-heap, persistence 가능) | 순수 메모리 기반, 매우 경량 |

## 4️⃣ 로컬 캐시 사용 시 주의사항 ⚠️

- 노드 간 **캐시 일관성**이 보장되지 않는다.
  (서버 다중 인스턴스 사용 시 유효성 문제 발생 가능 → 필요 시 Redis 등 분산 캐시 활용)
- JVM 메모리 사용량 주의. 최대 크기/TTL을 반드시 설정할 것.
- 로컬 캐시는 다음과 같은 용도로 적합:
  - 코드 테이블, 마스터 데이터 등 변동이 적은 정보
  - 사용자 권한 정보 등 다중 조회 시 반복 사용되는 데이터

---

## 5️⃣ 구성 흐름 요약

```
Controller → Service → @Cacheable 메서드 호출
   ↓
CacheManager 확인 → JVM 내 Cache 확인 (EhCache 또는 Caffeine)
   ↓
(있음) → 캐시에서 값 반환
(없음) → 메서드 실행 → Cache에 저장 후 반환
```

---

## 결론

✅ EhCache: **전통적**, 디스크 지원, 다양한 옵션 → 엔터프라이즈 레거시 시스템에 강점
✅ Caffeine: **최신, 경량, 초고속**, 메모리 기반 → 고성능 API 서버에 적합
✅ 둘 다 Spring Cache abstraction으로 쉽게 통합 가능 (`@Cacheable`, `@CacheEvict`, `@CachePut` 사용 동일)

# 스프링 애플리케이션 성능 튜닝

스프링 부트 애플리케이션은 개발 생산성과 유연성이 뛰어나지만, 기본 설정으로만 운영할 경우 성능 측면에서 병목이나 비효율이 발생할 수 있다.
본 절에서는 **주요 튜닝 포인트**를 단계별로 설명하며, 실제 운영 환경에서 성능 최적화를 달성하기 위한 전략을 제시한다.

---

## 1️⃣ 애플리케이션 프로파일 관리

- 운영 환경 별로 **profile**을 구분하여 최적화된 설정 적용.

```yaml
spring:
  profiles:
    active: prod
```

운영 환경에서는 다음 설정을 profile 별로 조정:

- connection pool 크기
- cache 설정
- 로그 레벨 (최소화)
- GC 정책

- metrics/monitoring 설정

---

## 2️⃣ 데이터베이스 연결 최적화

### Connection Pool 튜닝

Spring Boot는 기본적으로 **HikariCP**를 사용한다.
HikariCP는 고성능 connection pool이며, tuning 시 핵심 포인트는 다음과 같다:

```
spring:
  datasource:
    hikari:
      maximum-pool-size: 30
      minimum-idle: 10
      idle-timeout: 30000
      connection-timeout: 2000
      max-lifetime: 1800000
```

- `maximum-pool-size`: 최대 커넥션 개수
- `connection-timeout`: 커넥션 획득 시 최대 대기 시간

⚠️ 과도한 커넥션 개수 설정 시 DB 자원 고갈 우려가 있음. DB와 애플리케이션의 처리량 기준으로 적정 값 산정 필요.

### SQL 성능 모니터링

- Hibernate `show_sql`, `format_sql`, `generate_statistics` 활용
- Slow Query Log 활성화 (MySQL, PostgreSQL 등)
- **Query Plan** 분석 (EXPLAIN, ANALYZE)

### N+1 문제 해결

- `fetch join` 사용
- `@EntityGraph` 활용
- 필요 시 QueryDSL, Native Query 설계

---

## 3️⃣ 캐싱 전략 적용 🚀

효율적인 캐싱 적용은 성능 향상에 매우 효과적이다.

### 적용 대상 예시

- 자주 조회되는 참조 데이터 (코드 테이블 등)
- 복잡한 집계 결과
- 외부 API 호출 결과

### 기술 선택

- Caffeine → 고속 로컬 캐시
- Redis → 분산 캐시
- Hybrid 구성 (Caffeine + Redis)

### 적용 방법

```
@Cacheable(value = "commonCodes", key = "#codeId")
public CommonCode findCommonCode(Long codeId) {
    return commonCodeRepository.findById(codeId).orElse(null);
}
```

### 주의 사항

- 적절한 TTL 적용 필수
- 변경 빈도가 높은 데이터는 캐싱 대상에서 제외

---

## 4️⃣ HTTP 압축 및 헤더 최적화

### Gzip 압축 활성화

```
server:
  compression:
    enabled: true
    mime-types: application/json,application/xml,text/html,text/xml,text/plain
    min-response-size: 1024
```

- **네트워크 트래픽 절감** → 응답 속도 향상 🚀

### ETag 활성화

```
spring:
  web:
    resources:
      cache:
        use-etag: true
```

- 정적 리소스 캐싱 최적화 → 브라우저 캐시 활용

---

## 5️⃣ 빈 초기화 최적화

- **@Lazy** 사용으로 필요 시 로딩

```java
@Bean
@Lazy
public HeavyComponent heavyComponent() {
    return new HeavyComponent();
}
```

- 애플리케이션 시작 속도 개선

---

## 6️⃣ 비동기 처리 및 병렬화

### Async 적용

```java
@EnableAsync
@Service
public class AsyncService {

    @Async
    public void processHeavyTask() {
        // Heavy Logic
    }
}
```

- CPU 바운드 작업이나 I/O 작업을 병렬화하여 처리량 증가

### Thread Pool 튜닝

```java
@Configuration
@EnableAsync
public class AsyncConfig implements AsyncConfigurer {

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(10);
        executor.setMaxPoolSize(50);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("Async-Executor-");
        executor.initialize();
        return executor;
    }
}
```

---

## 7️⃣ GC 튜닝 및 JVM 옵션 설정 ☕

적절한 GC 설정은 **GC Pause Time**과 Throughput에 큰 영향을 준다.

운영 환경에서는 다음과 같은 JVM 옵션 튜닝이 고려된다:

```
-XX:+UseG1GC
-XX:MaxGCPauseMillis=200
-XX:+ParallelRefProcEnabled
-XX:+UnlockExperimentalVMOptions
-XX:+UseStringDeduplication
```

- G1GC 사용 → 안정적인 Pause Time 제공
- String Deduplication → 메모리 사용량 최적화

---

## 8️⃣ 모니터링 및 프로파일링 🔍

### Spring Boot Actuator 활용

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

- `/actuator/metrics`, `/actuator/health`, `/actuator/trace`

### 외부 모니터링 도구

- Prometheus + Grafana
- ELK Stack (Elasticsearch + Logstash + Kibana)
- Zipkin / Jaeger → 분산 추적

### 프로파일링 도구

- JFR (Java Flight Recorder)
- VisualVM
- YourKit

---

## 9️⃣ 기타 고려 사항

### Index 최적화

- 모든 Join / Where 절에 사용되는 컬럼에 Index 구성 여부 확인
- Index 과다 사용은 INSERT/UPDATE 성능 저하 유발 → 적절한 균형 필요

### API 응답 크기 최적화

- 필요 필드만 반환 (DTO 활용)
- @JsonIgnore 사용으로 불필요한 필드 제거

### HTTP Connection 재사용

- RestTemplate / WebClient 사용 시 Connection Pool 구성

---

## 종합 정리 🗂️

| 튜닝 영역 | 주요 기법 |
|---|---|
| DB 연결 | HikariCP 설정, 쿼리 최적화 |
| 캐싱 | Caffeine, Redis 적용 |
| HTTP 성능 | Gzip, ETag 활성화 |
| 빈 초기화 | @Lazy 적용 |
| 비동기 처리 | Async + ThreadPool 튜닝 |
| JVM | GC 튜닝, String Deduplication |
| 모니터링 | Actuator, Prometheus, Zipkin |
| API 응답 | DTO 활용, 필드 최소화 |

---

**성능 튜닝은 일회성 작업이 아니라 지속적인 모니터링과 최적화의 과정**이다.
애플리케이션의 특성과 트래픽 패턴에 맞는 튜닝 전략을 수립하고 주기적으로 검증하는 것이 중요하다. 🚀

# HTTP 압축, ETag 설정

스프링 부트 기반의 웹 애플리케이션에서는 네트워크 비용 최적화와 응답 성능 향상을 위해 **HTTP 압축**과 **ETag** 활용이 매우 효과적이다.
이 두 가지는 **전송 효율성**과 **캐시 최적화** 측면에서 주요한 튜닝 포인트로 활용된다.

---

## 1️⃣ HTTP 압축 (Gzip)

### 개요

HTTP 응답 본문을 Gzip 등의 알고리즘으로 압축하면 다음과 같은 효과가 있다:

- 네트워크 대역폭 절감
- 페이지 로드 속도 향상
- 모바일 환경에서 체감 성능 개선 📱

### 스프링 부트 설정

Spring Boot에서는 `application.yml` 또는 `application.properties`에서 간단하게 HTTP 압축을 설정할 수 있다.

### application.yml 예시

```
server:
  compression:
    enabled: true
    mime-types:
application/json,application/xml,text/html,text/xml,text/plain,text/css,text/javascript,
application/javascript
    min-response-size: 1024
```

### 주요 속성 설명

| 속성명 | 설명 |
| --- | --- |
| `enabled` | 압축 활성화 여부 |
| `mime-types` | 압축 대상 MIME 타입 목록 |
| `min-response-size` | 지정 크기 이상인 경우에만 압축 적용 (byte 단위) |

### 동작 흐름

```
클라이언트 → 요청 시 `Accept-Encoding: gzip` 헤더 포함
서버 → 설정된 MIME 타입 및 크기 조건 충족 시 Gzip 압축된 응답 전송
클라이언트 → `Content-Encoding: gzip` 응답 헤더 확인 후 압축 해제
```

### 검증 방법

브라우저 개발자 도구(Network 탭)에서 응답 헤더 확인:

```
Content-Encoding: gzip
```

👉 설정이 적용되었음을 확인 가능.

---

## 2️⃣ ETag 설정 🏷️

### 개요

ETag(Entity Tag)는 HTTP 프로토콜에서 제공하는 **리소스 식별자**이다.
리소스의 버전을 고유하게 식별할 수 있으며, 클라이언트와 서버 간 **효율적인 캐시 관리**에 활용된다.

활용 효과:

- 동일 리소스 재요청 시 변경 여부 확인 가능
- 불필요한 데이터 재전송 방지 → 네트워크 사용량 절감
- 정적 리소스(이미지, CSS, JS 등)에 특히 유용

## 동작 원리

```
1️⃣ 최초 요청
→ 서버는 응답 시 ETag 헤더 추가 (e.g., ETag: "34a64df551429fcc55e4d42a148795d9f25f89d4")

2️⃣ 이후 요청 시
→ 클라이언트는 If-None-Match: "..." 헤더 포함하여 요청

3️⃣ 서버는 ETag 비교 후
→ 변경 없음 → 304 Not Modified 응답 (본문 없음)
→ 변경됨 → 새로운 ETag와 함께 정상 응답
```

## 스프링 부트 설정

Spring Boot 2.x 이상에서는 **정적 리소스에 기본적으로 ETag 지원**을 제공한다.

## application.yml 예시

```yaml
spring:
  web:
    resources:
      cache:
        use-etag: true
```

또는 `application.properties` 사용 시:

```
spring.web.resources.cache.use-etag=true
```

## ETag 적용 대상

- 기본적으로 **정적 리소스**(Static Resources) 대상:
  - `/static`, `/public`, `/resources`, `/META-INF/resources` 내 정적 리소스
- Controller 레이어의 동적 컨텐츠에는 별도 필터 구현 필요

## 검증 방법

응답 헤더 확인:

```
ETag: "34a64df551429fcc55e4d42a148795d9f25f89d4"
```

후속 요청 시:

```
If-None-Match: "34a64df551429fcc55e4d42a148795d9f25f89d4"
```

응답:

```
304 Not Modified
```

👉 데이터 재전송 없이 상태 코드만 전송 → 트래픽 절감 🧊

## 3️⃣ HTTP 압축과 ETag의 관계

- HTTP 압축은 **응답 본문 크기**를 줄임.
- ETag는 **캐시 적중 여부**를 판단.

둘은 **서로 보완적인 역할**을 하며 동시에 적용 가능하다:

- 변경 없는 리소스는 **304 응답(ETag)** 으로 빠르게 처리.
- 변경된 리소스는 **압축 적용(Gzip)** 후 전송하여 네트워크 비용 최소화.

최적화된 워크플로우:

```
클라이언트 요청 → 서버에서 ETag 비교 → 변경 없음 → 304 응답
변경됨 → 압축 적용된 새 응답 전송
```

---

## 종합 정리 🗂️

| 항목 | 기능 | 기대 효과 |
| --- | --- | --- |
| HTTP 압축 (Gzip) | 응답 본문 압축 | 네트워크 대역폭 절감, 응답 속도 향상 |
| ETag | 리소스 버전 관리 및 캐싱 | 불필요한 데이터 재전송 방지, 네트워크 사용량 절감 |

---

## 적용 시 주의사항 ⚠️

- **동적 컨텐츠의 경우 ETag 자동 적용되지 않음** → 필요 시 커스텀 필터 구현 필요.
- HTTP 압축은 **CPU 사용량 증가** 가능성 존재 → 트래픽 규모와 서버 성능 고려 후 적용.
- ETag 적용 시 **리소스 해시 또는 LastModified 기반** 전략 명확히 수립.