18. 캐시 및 성능 최적화

Spring Cache 추상화

- `@Cacheable`, `@CacheEvict`, `@CachePut`

Spring에서는 **메서드 수준 캐시 기능**을 매우 간편하게 구현할 수 있도록 `@Cacheable`, `@CacheEvict`, `@CachePut`을 제공한다.
이 기능은 Spring Cache 추상화 API 위에서 동작하고, 실제 저장소로는 **EhCache, Redis, Caffeine 등**을 선택할 수 있다.

---

1. @Cacheable

🔹 개념

해당 메서드가 호출될 때, 파라미터 기반으로 캐시에 먼저 데이터를 조회하고, **존재하지 않을 경우에만 메서드를 실행해서 결과를 캐시에 저장**한다.

```
@Cacheable(value = "users", key = "#id")
public User findUserById(Long id) {
    return userRepository.findById(id).orElse(null);
}
```

🔹 작동 순서

1. `value`(=cache name)가 "users"인 캐시에서 `key = id`에 해당하는 데이터 찾기
2. 있으면 캐시된 값을 반환 (메서드 실행 ❌)
3. 없으면 메서드 실행 → 반환 결과를 캐시에 저장

🔹 주요 속성

속성	설명
`value`	캐시 이름 (필수)
`key`	SpEL 기반 키 지정 (`#id`, `#user.name` 등)
`condition`	특정 조건에만 캐시 적용
`unless`	조건이 true일 경우 캐시 ❌
`sync`	동기화 캐시 사용 (동시성 문제 대응)

예시 - 조건 설정

```
@Cacheable(value = "products", key = "#id", condition = "#id > 10", unless =
"#result == null")
public Product getProduct(Long id) { ... }
```

2. @CacheEvict

🔹 개념

메서드 실행 후, **지정된 캐시 항목을 제거**하는 역할을 해. 주로 **데이터 삭제/수정 시** 사용함.

```
@CacheEvict(value = "users", key = "#id")
public void deleteUser(Long id) {
    userRepository.deleteById(id);
}
```

🔹 주요 속성

속성	설명
`value`	캐시 이름
`key`	제거할 키 지정
`allEntries`	`true`이면 해당 캐시 전체 삭제
`beforeInvocation`	`true`면 메서드 실행 전에 캐시 제거 (기본은 후)

예시 - 전체 캐시 제거

```
@CacheEvict(value = "users", allEntries = true)
public void deleteAllUsers() {
    userRepository.deleteAll();
}
```

예시 - 예외 발생 전 제거

```
@CacheEvict(value = "users", key = "#id", beforeInvocation = true)
public void riskyDelete(Long id) {
    if (id == null) throw new IllegalArgumentException();
    userRepository.deleteById(id);
}
```

3. @CachePut

개념

항상 메서드를 실행하고, **그 결과를 캐시에 강제로 갱신(update)**한다.
즉, `@Cacheable`과 달리 항상 실행됨.

```
@CachePut(value = "users", key = "#user.id")
public User updateUser(User user) {
    return userRepository.save(user);
}
```

주요 속성

- `value`: 캐시 이름
- `key`: SpEL로 캐시 키 설정

> 갱신 대상 메서드가 실제 데이터를 변경할 때 유용

---

4. 예시: 세 애너테이션 통합 활용

```
@Service
public class UserService {

    @Cacheable(value = "users", key = "#id")
    public User findById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    @CachePut(value = "users", key = "#user.id")
    public User updateUser(User user) {
        return userRepository.save(user);
    }

    @CacheEvict(value = "users", key = "#id")
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

---

5. 캐시 키 관리 전략

Spring의 기본 키 생성 방식은 파라미터 기반이며, 커스텀 키 생성기를 설정할 수도 있음.

```java
@Bean
public KeyGenerator customKeyGenerator() {
    return (target, method, params) -> {
        return method.getName() + "_" + Arrays.toString(params);
    };
}
```

```java
@Cacheable(value = "users", keyGenerator = "customKeyGenerator")
```

6. 주의사항 및 팁

항목	주의 사항
프록시 기반	`@Cache*` 애너테이션은 프록시 방식이므로 **자기 클래스 내부 호출 시 동작하지 않음**
반환값 필수	`@Cacheable`, `@CachePut` 메서드는 반드시 반환값이 있어야 함
`null` 캐시	캐시 저장소에 따라 `null` 값 저장 여부 다름 (Redis는 기본적으로 저장 안 함)
트랜잭션	캐시는 DB 트랜잭션과 별개로 동작하므로 **트랜잭션 커밋 전 캐시 저장 주의**
테스트 시 주의	테스트 중엔 캐시를 수동으로 초기화하거나 꺼야 예외적인 케이스 확인 가능

7. 저장소 예시: Redis 연동

의존성 추가 (Gradle)

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

설정 예시 (`application.yml`)

```yaml
spring:
  cache:
    type: redis
  redis:
    host: localhost
    port: 6379
```

정리 요약

애너테이션	역할	호출 여부	캐시 영향
`@Cacheable`	캐시된 결과 있으면 재사용	조건부 실행	조회
`@CacheEvict`	캐시 제거	항상 실행	삭제
`@CachePut`	결과를 항상 캐시에 갱신	항상 실행	갱신

캐시 저장소: Redis, Caffeine

🔰 공통 전제: Spring Cache 추상화

Spring은 자체 `@Cacheable`, `@CacheEvict`, `@CachePut` 등을 사용해 **표준 API를 제공**하고, 실제 저장소 구현체를 자유롭게 바꿀 수 있다.

```
          ┌─────────────┐
          |  @Cacheable|
          └─────┬───────┘
                ↓
      [Spring Cache 추상화]
                ↓
┌──────────┬─────────────┐
|   Redis  | Caffeine |
└──────────┴─────────────┘
```

---

1. ✅ Redis 캐시 저장소

🔹 특징

항목	설명
분산	Yes (다중 서버 가능)
영속성	Yes (옵션)
TTL 지원	Yes
크기 제한	없음 (메모리 기반)
속도	빠름 (네트워크 비용은 존재)
장점	다중 서버 간 캐시 공유 가능
단점	외부 서비스 의존, Redis 설치 필요

🔹 Gradle 의존성

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

🔹 application.yml 설정 예시

```yaml
spring:
  cache:
    type: redis
  redis:
    host: localhost
    port: 6379
```

🔹 TTL 및 캐시 구성 설정 (Java 기반)

```java
@Configuration
public class RedisCacheConfig {

    @Bean
    public RedisCacheConfiguration redisCacheConfiguration() {
        return RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(10)) // 기본 TTL 설정
            .disableCachingNullValues();     // null 캐시 방지
    }
}
```

🔹 TTL을 캐시마다 다르게 설정 (예: CacheManager customizer)

```java
@Bean
public CacheManager cacheManager(RedisConnectionFactory connectionFactory) {
    Map<String, RedisCacheConfiguration> configMap = new HashMap<>();
    configMap.put("shortCache",
RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofSeconds(30)));
    configMap.put("longCache",
RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofHours(1)));

    return RedisCacheManager.builder(connectionFactory)
        .withInitialCacheConfigurations(configMap)
        .build();
}
```

---

2. ✅ Caffeine 캐시 저장소 (내장형)

🔹 특징

항목	설명
분산	❌ (단일 인스턴스 전용)
영속성	❌ (JVM 종료 시 소멸)
TTL 지원	Yes

항목	설명
LRU 지원	Yes
속도	매우 빠름 (메모리 내)
장점	설치 불필요, 성능 탁월
단점	분산 캐시 불가능

🔹 Gradle 의존성

```
implementation 'org.springframework.boot:spring-boot-starter-cache'
implementation 'com.github.ben-manes.caffeine:caffeine'
```

🔹 application.yml 설정 예시

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

> `maximumSize`: 최대 저장 개수
> `expireAfterWrite`: 쓰기 이후 만료 시간

🔹 Java 기반 구성 (선택 사항)

```
@Bean
public Caffeine<Object, Object> caffeineConfig() {
    return Caffeine.newBuilder()
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .maximumSize(1000);
}

@Bean
public CacheManager cacheManager(Caffeine<Object, Object> caffeine) {
    return new CaffeineCacheManager("users", "products");
}
```

---

3. Redis vs Caffeine 비교 정리

항목	Redis	Caffeine
설치 필요	O (서버 별도)	X

항목	Redis	Caffeine
분산 캐시	O (다중 서버 공유)	X
속도	빠름 (네트워크 존재)	더 빠름 (로컬 JVM)
TTL	지원	지원
LRU/최대 크기	설정 가능	기본 지원
재시작 후 보존	가능 (RDB, AOF)	불가능
캐시 공유	클러스터/마이크로서비스용	단일 서비스 전용
운영 난이도	높음	낮음

---

4. 캐시 이름 관리 팁

- `@Cacheable(value = "users")`에서 `"users"`는 반드시 `CacheManager`가 인식하는 이름이어야 함
- 동적으로 캐시 생성하려면 `SimpleCacheManager` 또는 `CaffeineCacheManager`/`RedisCacheManager` 설정 필요

---

5. 실전 사용 전략

시나리오	권장 캐시
단일 애플리케이션	Caffeine
멀티 인스턴스 배포	Redis
빠른 TTL 회전 필요	Caffeine
사용자 인증 정보 저장	Redis (다중 인스턴스용)
실시간 API 응답 캐싱	Redis (단, 높은 부하 시 Caffeine도 고려)
임시 계산 캐시	Caffeine
사용자별 데이터 (세션, 권한)	Redis + Spring Session

✅ 다음으로?

- Redis + Caffeine을 **혼합 구성**할 수도 있어: **로컬 우선 캐시(Caffeine)** → 실패 시 Redis → 실패 시 DB
- `Spring CacheManager`를 **멀티 계층 캐시(Multi-tier)**로 커스터마이징하는 것도 가능

TTL, LRU 전략 설정

1. TTL (Time To Live)

🔹 개념

- TTL은 **데이터의 유효 기간**을 설정하여, 일정 시간이 지나면 자동으로 캐시에서 삭제되게 함
- 메모리 누수 방지, 오래된 데이터 제거에 필수

✅ Redis에서 TTL 설정

방법 1: Global TTL 설정 (전체 캐시에 일괄 적용)

```
@Bean
public RedisCacheConfiguration redisCacheConfiguration() {
    return RedisCacheConfiguration.defaultCacheConfig()
        .entryTtl(Duration.ofMinutes(10));  // 모든 캐시에 10분 TTL
}
```

방법 2: 캐시별 TTL 설정 (캐시 이름마다 TTL 다르게)

```
@Bean
public CacheManager cacheManager(RedisConnectionFactory connectionFactory) {
    Map<String, RedisCacheConfiguration> configMap = new HashMap<>();
    configMap.put("shortCache", RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofSeconds(30)));
    configMap.put("longCache", RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofHours(1)));

    return RedisCacheManager.builder(connectionFactory)
        .withInitialCacheConfigurations(configMap)
        .cacheDefaults(RedisCacheConfiguration.defaultCacheConfig().entryTtl(Duration.ofMinutes(10)))
        .build();
}
```

✅ Caffeine에서 TTL 설정

```
spring:
  cache:
    type: caffeine
  caffeine:
    spec: expireAfterWrite=5m,expireAfterAccess=10m
```

또는 Java 기반 설정:

```java
@Bean
public Caffeine<Object, Object> caffeineConfig() {
    return Caffeine.newBuilder()
        .expireAfterWrite(5, TimeUnit.MINUTES)
        .expireAfterAccess(10, TimeUnit.MINUTES);
}
```

전략	설명
`expireAfterWrite`	쓰기 이후 경과 시간 기준으로 TTL 적용
`expireAfterAccess`	마지막 접근 시점부터 TTL 적용 (읽기 포함)

---

2. LRU (Least Recently Used)

🔹 개념

- LRU는 **가장 오래 사용되지 않은 캐시 항목을 제거**하는 방식
- 캐시 용량 초과 시 메모리 보호용으로 가장 많이 쓰임

✅ Redis에서 LRU 설정

Redis는 TTL과는 다르게, **LRU 정책은 Redis 서버 자체 설정**에서 수행됨.

① Redis 설정파일 (redis.conf)

```
maxmemory 256mb
maxmemory-policy allkeys-lru
```

② 메모리 설정 명령어로도 가능

```
CONFIG SET maxmemory 256mb
CONFIG SET maxmemory-policy allkeys-lru
```

정책	설명
`noeviction`	기본값, 메모리 초과 시 오류 발생
`allkeys-lru`	전체 키 대상 LRU 제거
`volatile-lru`	TTL 설정된 키만 대상으로 LRU 제거
`allkeys-random`	무작위 제거
`volatile-ttl`	TTL이 가장 가까운 키 제거

✅ Caffeine에서 LRU 유사 정책 설정

Caffeine은 내부적으로 LRU보다 더 정교한 **Window TinyLFU (W-TinyLFU)** 알고리즘을 사용하지만, 일반적인 LRU와 거의 동일한 효과를 제공한다.

```
spring:
  cache:
    type: caffeine
  caffeine:
    spec: maximumSize=1000
```

또는 Java 설정:

```
@Bean
public Caffeine<Object, Object> caffeineConfig() {
    return Caffeine.newBuilder()
        .maximumSize(1000) // 초과 시 LRU 방식으로 제거
        .expireAfterAccess(10, TimeUnit.MINUTES);
}
```

> `maximumSize`: 항목 수 제한
> 내부적으로 LRU와 LFU 혼합 전략 (빈도 + 최근성)

---

3. TTL + LRU 동시 설정 전략

Redis: TTL + 서버 레벨 LRU 조합

- `@Cacheable` 결과는 TTL로 만료되며,
- Redis 전체 메모리가 초과되면 **LRU 기준으로 자동 제거됨**

Caffeine: TTL + 최대 크기 조합

```
return Caffeine.newBuilder()
    .maximumSize(1000)             // LRU 제거
    .expireAfterWrite(10, TimeUnit.MINUTES) // TTL
    .build();
```

---

4. 전략 선택 팁

상황	권장 설정
캐시 항목 수 제한 필요	`maximumSize` (Caffeine), `maxmemory-policy` (Redis)
민감한 데이터 자동 만료	`expireAfterWrite` (Caffeine), `entryTtl` (Redis)
최근 사용 위주로 보존	`expireAfterAccess`, `allkeys-lru`
전역 설정 + 세부 커스터마이징	Redis + 캐시별 TTL

상황	권장 설정
캐시 부담 없는 빠른 처리	Caffeine (로컬 메모리 빠름)

---

5. 테스트 및 검증 방법

Redis에서 TTL 확인

```
TTL users::123
```

→ 결과가 `300`이면 5분 남았다는 뜻

Caffeine은 JVM 내 확인 필요 → 로그 출력 또는 `MeterRegistry` + Actuator 이용

✅ 마무리 요약

항목	Redis	Caffeine
TTL 설정	`entryTtl(Duration)`	`expireAfterWrite()`
LRU 설정	`maxmemory-policy`	`maximumSize()`
강제 삭제	`@CacheEvict`	동일
복합 전략	TTL + LRU 가능	TTL + Size 가능

동시성 고려한 캐시 업데이트

1. 문제 유형 요약

문제 이름	설명
❗ Cache Stampede	다수의 요청이 동시에 캐시 미스를 발생시키며 DB로 몰림
❗ Race Condition	동시에 여러 스레드가 동일 데이터를 캐시에 쓰며 결과가 꼬임
❗ Inconsistency	DB는 업데이트되었지만 캐시는 여전히 예전 값을 들고 있음
❗ Thundering Herd	하나의 키에 많은 트래픽이 몰릴 때 TTL 만료 직후 요청이 폭주

---

2. 해결 전략 개요

전략	설명
✅ 동기화(Cache Lock)	특정 키에 대해 동시에 하나만 갱신하도록 Lock 사용
✅ Refresh-Ahead	만료 전에 백그라운드에서 미리 갱신

전략	설명
✅ Local Lock + Redis Lock 혼합	JVM 내부 + 분산 환경 고려
✅ Double Check	캐시 확인 → DB 확인 → 다시 캐시 확인
✅ Async 갱신	캐시 갱신을 비동기 작업으로 분리

---

3. ✅ 방법 1: `@Cacheable(sync = true)` 사용

> Spring 4.3+ 지원
> `sync = true`를 사용하면 **동일 키로 동시에 들어온 요청은 하나만 실행**되고 나머지는 기다린다.

```
@Cacheable(value = "products", key = "#id", sync = true)
public Product getProduct(Long id) {
    return productRepository.findById(id).orElseThrow();
}
```

⚠️ 제약사항

- 동기화는 **개별 캐시 항목(key) 수준**임
- 단일 JVM에서만 작동 (분산 환경은 별도 처리 필요)

---

4. ✅ 방법 2: 분산 락 (예: Redis Lock)

```
String lockKey = "lock:product:" + id;
boolean acquired = redisTemplate.opsForValue().setIfAbsent(lockKey, "1",
Duration.ofSeconds(5));

if (acquired) {
    try {
        Product data = repository.findById(id).orElseThrow();
        redisTemplate.opsForValue().set("product:" + id, data, Duration.ofMinutes(5));
        return data;
    } finally {
        redisTemplate.delete(lockKey); // 락 해제
    }
} else {
    // 락이 걸려 있으면 대기하거나 fallback
    Thread.sleep(100); return getFromCache(); // 재시도
}
```

라이브러리 활용 예시

- Redisson
- Lettuce 기반 Redis Lock

---

5. ✅ 방법 3: Caffeine Refresh Ahead (비동기 미리 갱신)

> Caffeine은 TTL이 도달하기 전에 백그라운드 스레드에서 데이터를 미리 갱신할 수 있음

```
return Caffeine.newBuilder()
    .maximumSize(1000)
    .refreshAfterWrite(10, TimeUnit.MINUTES)
    .build(key -> loadFromDatabase(key));
```

⚠️ 주의

- `refreshAfterWrite`는 **데이터가 캐시된 후 특정 시간이 지나면 자동으로 비동기 갱신**
- 갱신 함수는 `build(Function<K, V>)`를 통해 등록해야 함

---

6. ✅ 방법 4: Double Check + Fallback

```
public Product getProduct(Long id) {
    Product cached = cache.get(id);
    if (cached != null) return cached;

    synchronized (("productLock:" + id).intern()) {
        // Double check
        Product recheck = cache.get(id);
        if (recheck != null) return recheck;

        Product dbData = repository.findById(id).orElseThrow();
        cache.put(id, dbData);
        return dbData;
    }
}
```

- `String.intern()`을 통해 JVM 내에서 동기화
- 또는 `ConcurrentHashMap<id, ReentrantLock>` 방식도 가능

---

7. ✅ 방법 5: Cache Aside 패턴 with 갱신 이벤트

```
// 1단계: DB를 갱신한다
repository.update(product);

// 2단계: 캐시를 무효화하거나 직접 덮어쓴다
cacheManager.getCache("products").evict(product.getId()); // 또는 put()
```

- 캐시를 **완전히 비우거나 최신 데이터로 overwrite**
- 캐시 무효화 시점은 트랜잭션 커밋 이후가 안전함

---

8. 전략 비교 정리

전략	동시성 제어	분산 환경	구현 복잡도	실용성
`@Cacheable(sync = true)`	✅	❌	아주 낮음	단일 인스턴스 앱
Redis Lock	✅	✅	중간	마이크로서비스
Caffeine `refreshAfterWrite`	⭕ (비동기)	❌	낮음	빠른 로컬 캐시
Double Check	✅	❌	낮음	로컬 앱
Cache Aside	✅ (제어 가능)	✅	중간	범용

---

9. 실전 시나리오 예시

```
사용자가 상품 상세 정보를 요청함
→ 캐시 미스 발생
→ 하나의 요청이 DB에서 데이터를 가져옴 (Lock 획득)
→ 결과를 캐시에 저장
→ 다른 요청은 캐시에서 읽음
→ 이후 5분 지나면 자동 만료 또는 refresh
```

✅ 마무리 정리

- 캐시 동시성 문제는 단순히 `@Cacheable`만으로는 완벽히 해결되지 않음
- `sync = true`는 간단하지만 **단일 서버 한정**
- **Redis Lock + Double Check**가 마이크로서비스 환경에서 가장 널리 쓰임
- Caffeine은 **refreshAhead** 기능을 활용하면 뛰어난 성능
- 데이터 일관성이 중요한 경우엔 항상 **트랜잭션과 캐시 업데이트 시점**을 주의

API 응답 캐싱

1. 개요: 왜 API 응답을 캐싱할까?

API 응답은 일반적으로 다음과 같은 경우에 캐싱이 유리해:

- 자주 호출되지만 변경이 드문 데이터
  (예: 상품 목록, 카테고리, 환율, 공공 API)
- 응답 생성 비용이 비싼 작업 (복잡한 JOIN, 외부 API 연동 등)
- 실시간성이 크게 중요하지 않고, 약간의 지연 허용 가능할 때

---

2. 기본 방식: `@Cacheable`로 응답 캐싱

Spring에서는 **컨트롤러나 서비스 레이어 메서드**에 `@Cacheable`을 붙이는 것만으로 캐싱 가능

```
@Cacheable(value = "productCache", key = "#id")
public ProductResponse getProductById(Long id) {
    log.info("DB or 외부 호출 수행");
    return productService.getProduct(id);
}
```

이때 캐시에 저장되는 것은?

- **메서드의 리턴값** 전체가 캐시됨
- 직렬화 가능해야 함 (Redis는 Jackson 기반으로 자동 직렬화)

---

3. Controller 레벨 캐싱도 가능할까?

직접은 안 되지만, **서비스 레이어에서 처리**하는 것이 권장됨. 하지만 예외적으로 Controller에도 적용 가능:

```
@RestController
@RequiredArgsConstructor
public class ProductController {
    private final ProductService productService;

    @GetMapping("/api/product/{id}")
    public ResponseEntity<ProductResponse> getProduct(@PathVariable Long id) {
        return ResponseEntity.ok(productService.getProductById(id)); // 서비스에서 캐싱
    }
}
```

---

4. 캐시 Key 전략

전략	예시	설명
단일 파라미터	`key = "#id"`	숫자, ID
복수 파라미터	`key = "#name + '_' + #type"`	문자열 조합
Request 객체 전체	`key = "#request.toString()"`	권장 ❌, 명확한 키 생성기 사용 추천

커스텀 키 생성기

```
@Bean
public KeyGenerator keyGenerator() {
    return (target, method, params) ->
        method.getName() + "_" + Arrays.toString(params);
}
```

```
@Cacheable(value = "myApiCache", keyGenerator = "keyGenerator")
```

---

5. TTL 설정과 캐시 저장소 지정

Redis를 사용하는 경우

```
spring:
  cache:
    type: redis
  redis:
    host: localhost
    port: 6379
```

```
@Bean
public RedisCacheConfiguration redisCacheConfiguration() {
    return RedisCacheConfiguration.defaultCacheConfig()
        .entryTtl(Duration.ofMinutes(3))
        .disableCachingNullValues();
}
```

---

6. 실전 패턴: API 응답 캐싱 패턴 예시

```
@Cacheable(value = "exchangeRates", key = "#currency", unless = "#result == null")
public ExchangeRateResponse getRate(String currency) {
    // 외부 환율 API 호출
}
```

```
@CacheEvict(value = "exchangeRates", key = "#currency")
public void refreshRate(String currency) {
    // 수동 캐시 갱신
}
```

---

7. 고급 전략: 조건부 캐싱

```
@Cacheable(value = "products", key = "#id", condition = "#id > 0", unless = "#result == null")
public ProductResponse getProductById(Long id) {
    ...
}
```

- `condition` : true일 때만 캐시 시도
- `unless` : true일 때 캐싱하지 않음 (캐시 무시)

---

8. 주의사항

항목	주의
객체 직렬화	Redis에 저장 시 직렬화 필요 (Jackson이 기본)
캐시 무효화	DB 변경 시 `@CacheEvict` 필수
캐시 범람	key 범위 제한 / TTL 필수
예외 발생 시	캐시에 저장하지 않도록 `unless = "#result == null"` 설정
실시간 응답	주기적 갱신 전략 필요 (예: `@Scheduled`)

---

9. 실전 설계 예시: 제품 상세 조회 API

```
@GetMapping("/api/products/{id}")
public ProductResponse getProduct(@PathVariable Long id) {
    return productService.getProductById(id); // 내부에서 캐시 사용
}
```

```
@Cacheable(value = "productDetails", key = "#id")
public ProductResponse getProductById(Long id) {
    log.info("DB 호출 발생");
    return repository.findProductDetail(id);
}
```

---

10. 캐시 무효화 전략

시점	방법
데이터 수정	`@CacheEvict(key=...)`
배치 처리	`cacheManager.getCache(...).clear()`
TTL 자동 만료	Redis: `entryTtl`, Caffeine: `expireAfterWrite`
강제 리로드	`@CachePut` 또는 API 수동 호출 후 캐시 갱신

---

11. 도구 및 보조 전략

도구	목적
Redis	분산 캐시 저장소
Caffeine	빠른 로컬 캐시

도구	목적
Actuator + /caches	현재 캐시 상태 확인
Spring AOP + TTL	비동기 자동 갱신 설계 가능
MeterRegistry	캐시 적중률 모니터링 가능 (Micrometer)

✅ 마무리 요약

항목	설명
목적	API 응답 속도 향상, DB 부하 감소
위치	서비스 계층에 `@Cacheable` 권장
TTL	Redis / Caffeine 모두 지원
무효화	`@CacheEvict`, TTL, 수동 방식
조건부 캐싱	`condition`, `unless`로 제어
저장소	Redis(멀티 인스턴스), Caffeine(로컬 고속)