

JPA

ORM(Object-Relational Mapping) 이란?

ORM은 객체-관계 매핑을 의미하며, 객체 지향 프로그래밍의 클래스와 관계형 데이터베이스의 테이블을 자동으로 매핑해주는 기술입니다.

ORM의 주요 개념

1. 객체와 테이블 매핑: 엔티티 클래스를 데이터베이스 테이블과 연결
2. 자동 SQL 생성: 개발자가 직접 SQL을 작성하지 않아도 데이터 조작 가능
3. 영속성 컨텍스트(Persistence Context): 객체의 상태를 관리하고 변경을 자동으로 반영
4. 트랜잭션 관리: ORM 프레임워크가 데이터 일관성을 유지

ORM의 장점

- ✓ SQL을 직접 작성할 필요 없이 코드만으로 데이터 조작 가능
- ✓ 데이터베이스 벤더(Oracle, MySQL 등) 독립적인 개발 가능
- ✓ 객체지향적인 코드 작성으로 유지보수 용이

ORM의 대표적인 프레임워크

- **JPA(Java Persistence API)** – Java 표준 ORM
- **Hibernate** – JPA의 대표적인 구현체
- **MyBatis** – SQL Mapper 기반의 ORM 대체 기술

1. JPA란?

- Java Persistence API의 약자로, 자바 애플리케이션에서 관계형 데이터베이스를 쉽게 다룰 수 있도록 제공되는 표준 API
 - ORM(Object-Relational Mapping) 기술을 기반으로 동작
 - JDBC를 직접 사용하지 않고도 객체를 데이터베이스 테이블과 매핑하여 사용 가능
-

2. JPA의 주요 특징

- 객체와 관계형 데이터베이스 매핑: 객체지향 모델을 그대로 유지하면서 관계형 데이터베이스와 매핑 가능
 - JPQL (Java Persistence Query Language): 객체를 대상으로 하는 쿼리 언어 제공
 - 트랜잭션 관리: 엔터프라이즈 환경에서 강력한 트랜잭션 처리 지원
 - 캐싱 (Caching): 1차 캐시(영속성 컨텍스트) 및 2차 캐시 지원
 - 자동 DDL 생성: Entity 클래스 정의만으로 테이블 자동 생성 가능
 - 벤더 독립성: Hibernate, EclipseLink, OpenJPA 등 다양한 구현체 사용 가능
-

3. JPA와 Hibernate

- JPA는 표준 인터페이스이며, Hibernate는 JPA의 대표적인 구현체
 - Hibernate는 JPA의 기능을 확장하여 보다 많은 기능 제공
 - JPA 인터페이스를 사용하면 벤더 종속성을 줄일 수 있음
-

Hibernate는 JPA의 대표적인 구현체로, Java 애플리케이션에서 관계형 데이터베이스와 객체를 매핑하는 ORM 프레임워크입니다.

Hibernate의 주요 특징

1. **JPA 표준 구현**: JPA의 모든 기능을 지원하며 추가적인 기능도 제공
2. **자동 SQL 생성 및 실행**: 엔티티 매핑을 기반으로 SQL을 자동으로 생성하여 실행
3. **Lazy & Eager Loading**: 연관된 엔티티를 지연(Lazy) 또는 즉시(Eager) 로딩할 수 있음
4. **캐싱 기능**: 1차 캐시(영속성 컨텍스트) 및 2차 캐시(외부 캐시 제공) 지원
5. **트랜잭션 관리**: 선언적 트랜잭션을 지원하여 데이터 일관성을 유지

Hibernate의 장점

- SQL 문을 직접 작성하지 않아도 데이터베이스 연동 가능
- 데이터베이스 변경 시 코드 수정이 최소화됨
- 다양한 관계형 데이터베이스를 지원하며 독립적인 개발 가능

Hibernate와 JPA의 차이점

- **JPA**는 Java EE의 공식 표준 API이며, **Hibernate**는 JPA의 구현체 중 하나
- Hibernate는 JPA 외에도 자체적인 확장 기능을 제공

4. 주요 개념 및 어노테이션

(1) 엔티티(Entity) 클래스

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 100)
    private String name;

    @Column(unique = true)
    private String email;
}
```

(2) Repository 인터페이스

```
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByName(String name);
}
```

(3) Service 레이어

```
@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }
}
```

5. JPA의 생명주기 (Entity Life Cycle)

1. **비영속 (New)**: JPA가 관리하지 않는 상태
 2. **영속 (Managed)**: `EntityManager.persist()` 를 호출하여 관리 상태로 변경됨
 3. **준영속 (Detached)**: `EntityManager.detach()` 또는 `close()` 가 호출된 상태
 4. **삭제 (Removed)**: `EntityManager.remove()` 를 호출하여 삭제됨
-

6. JPQL과 네이티브 쿼리

(1) JPQL 예제

```
@Query("SELECT u FROM User u WHERE u.name = :name")  
List<User> findByNameJPQL(@Param("name") String name);
```

(2) 네이티브 쿼리 예제

```
@Query(value = "SELECT * FROM users WHERE name = ?1", nativeQuery = true)  
List<User> findByNameNative(String name);
```

7. JPA의 장단점

(1) 장점

- 객체 지향적인 데이터베이스 접근 가능
- 반복적인 SQL 코드 감소
- 유지보수 용이
- 캐싱 기능 지원으로 성능 향상

(2) 단점

- 학습 곡선이 있음
- 복잡한 쿼리는 SQL보다 성능이 떨어질 수 있음
- 초반 설정이 다소 복잡할 수 있음

8. Spring Data JPA

- JPA를 더욱 쉽게 사용할 수 있도록 도와주는 Spring 기반의 기술
- `JpaRepository`를 활용하여 기본적인 CRUD 메서드 자동 제공
- 페이징 및 정렬 기능 지원

```
public interface UserRepository extends JpaRepository<User, Long> {  
    Page<User> findByNameContaining(String name, Pageable pageable)  
}
```

9. 결론

- JPA는 객체와 데이터베이스 간의 매핑을 도와주는 강력한 API
- Spring Data JPA와 함께 사용하면 더욱 쉽게 활용 가능
- 적절한 설정과 최적화를 통해 성능을 개선할 수 있음

성능 최적화


항목	사용 목적
AJAX	클라이언트 단 성능 최적화 / 빠른 UX
JPA 튜닝	서버 단 데이터 처리 성능 최적화
Redis	응답 속도 향상 / 서버 부하 분산
k6, JMeter	부하 테스트 / 응답시간 분석

1. fetch join → SQL JOIN

예시

java

 복사


 편집

```
@Query("SELECT p FROM Person p JOIN FETCH p.orders")  
List<Person> findAllWithOrders();
```

실제 SQL로 변환되면:

sql

 복사

 편집

```
SELECT *  
FROM person p  
JOIN orders o ON p.id = o.person_id
```


- 즉, JPA가 자동으로 JOIN SQL을 생성해서 한 번의 쿼리로 연관된 데이터를 모두 가져와요.
- 결과는 객체 그래프 형태로 자동 매핑됨 (Person → orders 리스트에)

2. @BatchSize → IN 절 사용

설정

java

 복사

 편집

```
@OneToMany(mappedBy = "person")
@BatchSize(size = 10)
private List<Order> orders;
```

실제 동작

- 여러 `Person` 을 먼저 조회한 다음,
- 그 `Person` 의 ID들을 모아서 **IN** 쿼리로 한 번에 **orders** 조회:

sql

 복사

 편집

```
SELECT * FROM orders WHERE person_id IN (1, 2, 3, ..., 10)
```

- 여러 쿼리를 ****배치 묶음(IN 절)****으로 줄이는 전략이에요.

정리 비교

항목	fetch join	@BatchSize
SQL 형태	JOIN	IN
장점	연관 데이터까지 한 번에 다 가져옴	N+1 문제 완화 (쿼리 수 감소)
단점	중복 결과 조심 (JOIN으로 인해)	쿼리 수는 줄지만 여러 번 나뉘어 질 수 있음
언제 사용?	바로 함께 필요한 데이터	지연 로딩(LAZY) 유지하고 싶을 때

✅ N+1 문제란?

엔티티 1개를 조회할 때 연관된 엔티티 N개를 각각 따로 조회해서, 총 N+1개의 쿼리가 발생하는 문제입니다.

📦 예시로 쉽게 이해하기

상황:

- 우리가 고객(Person) 목록을 가져오고,
- 각 고객의 주문(Order) 내역도 함께 보고 싶어요.

java

📄 복사

✎ 편집

```
List<Person> people = personRepository.findAll(); // 고객 전체 조회
for (Person person : people) {
    List<Order> orders = person.getOrders(); // 각 고객의 주문 가져오기
}
```

📌 문제 발생 과정:

1. 1개의 쿼리로 모든 `Person` 가져옴

sql

📄 복사

✎ 편집

```
SELECT * FROM person;
```

2. 그리고 각 **Person**의 **orders**를 따로따로 조회함 (예: 5명이라면 5번 더 조회)

sql

📄 복사

✎ 편집

```
SELECT * FROM orders WHERE person_id = 1;  
SELECT * FROM orders WHERE person_id = 2;  
SELECT * FROM orders WHERE person_id = 3;  
SELECT * FROM orders WHERE person_id = 4;  
SELECT * FROM orders WHERE person_id = 5;
```

👉 결과적으로 쿼리가 총 $1 + 5 = 6$ 번 실행됨

💣 왜 문제일까?

- 고객이 10명, 100명, 1000명이 되면?
- 매번 쿼리가 N개씩 증가해서 DB 부하 심각
- 성능 저하 및 대기 시간 증가

N+1 문제는 연관된 데이터를 각각 조회해서
생기는 과도한 쿼리 문제이고,
이를 해결하려면 **fetch join**이나 **batch size** 등을
활용해 **한 번에** 가져오는 방식으로 처리

✅ 해결 방법

해결 방법

설명

`fetch join`

한 번의 쿼리로 연관된 데이터도 같이 가져옴

`@EntityGraph`

연관 필드를 명시적으로 fetch join 하도록 지정

`@BatchSize`

여러 LAZY 필드를 한 번에 IN 쿼리로 가져오도록 설정

✅ 지연 로딩(LAZY)이란?

데이터를 "필요할 때까지" 가져오지 않고, 나중에 실제로 사용할 때 가져오는 방식이에요.

🍜 라면 비유로 쉽게 설명

- 배달 앱에서 라면을 주문한다고 생각해보세요.

✅ 즉시 로딩(EAGER):

- 주문하자마자 모든 재료(면, 스프, 그릇, 젓가락...)를 한 번에 다 배송함
- 지금 당장 안 쓸 재료도 다 받음 → 빠르지만 비효율적

✅ 지연 로딩(LAZY):

- 처음에는 라면만 받고,
- 스프나 그릇은 필요할 때 앱에서 추가로 주문해서 받음
→ 필요할 때만 가져오니까 효율적

실제 JPA 상황으로 예시

java

📄 복사

✎ 편집

```
@Entity
public class Person {
    @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
    private List<Order> orders;
}
```

java

📄 복사

✎ 편집

```
Person person = personRepository.findById(1L).get();
```

이때!

- DB에서는 **Person** 정보만 조회합니다.
- `orders` 는 아직 안 가져와요! (프록시 객체만 있음)

java

📄 복사

✎ 편집

```
List<Order> orders = person.getOrders(); // 이 시점에 DB에서 orders 쿼리 실행!
```

지연 로딩의 장점

장점	설명
성능 향상	처음에 필요 없는 데이터는 안 가져와서 빠름
메모리 절약	정말 필요한 시점에만 로딩됨
유연한 데이터 처리	필요한 곳에서만 연관 데이터 로딩 가능

로딩 방식	설명	쿼리 실행 시점
EAGER	연관 객체를 즉시 함께 로딩	엔티티 조회 시 바로
LAZY	연관 객체를 나중에 필요할 때 로딩	실제 접근 시점에

성능 개선 요점 요약

전략	설명
Entity → DTO 변환	필요한 필드만 전달, 직렬화 비용 줄임
Fetch Join 사용	LAZY 로딩으로 인한 N+1 문제 제거
Controller에서 Entity 직접 반환 금지	순환 참조, 프록시 객체 문제 방지

5. Api 응답 성능개선

1.캐싱 적용

- @Cacheable 어노테이션을 사용하여 각 엔드포인트의 응답을 캐싱
- Caffeine 캐시를 사용하여 메모리 기반 캐싱 구현
- 캐시 만료 시간 600초 설정

2.비동기 처리

- @Async 어노테이션을 사용하여 비동기 처리 구현
- CompletableFuture를 사용하여 비동기 응답 처리
- 스레드 풀 설정으로 동시성 제어

3.응답 압축

- GZIP 압축 활성화
- 주요 MIME 타입에 대한 압축 설정
- 최소 응답 크기 1024바이트 설정

4.서버 최적화

- 톰캣 스레드 풀 최적화
- HTTP/2 프로토콜 활성화
- 캐시 컨트롤 헤더 설정

5.스트리밍 응답

- 대용량 데이터 처리를 위한 스트리밍 응답 구현
 - StreamingResponseBody를 사용하여 청크 단위 전송
- 추가로 필요한 의존성을 pom.xml에 추가해야 합니다. Caffeine 캐시를 사용하기 위해 다음 의존성을 추가

ResponseEntity

java

복사

편집

```
return ResponseEntity.ok()  
    .headers(headers)  
    .body(stream);
```

이 한 줄은 다음 세 가지를 의미합니다:

1. **HTTP 상태 코드**: 200 OK 설정
2. **응답 헤더 설정**: 파일 타입, 인코딩, 파일 이름 등 지정
3. **응답 바디**: 실시간 생성한 스트리밍 데이터 (`StreamingResponseBody`)



1. `ResponseEntity.ok()`

java

📄 복사

✎ 편집

```
ResponseEntity.ok()
```

- **HTTP 200 OK** 상태코드를 반환하는 `ResponseEntityBuilder` 를 생성
- 이후 `.headers()`, `.body()` 등 메서드를 체이닝 방식으로 이어서 사용할 수 있음



2. `.headers(headers)`

java

📄 복사

✎ 편집

```
.headers(headers)
```

- 사용자가 직접 설정한 `HttpHeaders` 객체를 응답에 포함시킴

1) 캐싱 적용

Caffeine: 고성능 캐시(Cache) 를 구현할 때 사용하는 Java 기반 라이브러리

```
<dependency>  
  <groupId>com.github.ben-manes.caffeine</groupId>  
  <artifactId>caffeine</artifactId>  
  <version>3.1.8</version>  
</dependency>
```

✅ 작동 원리

```
java  
  
@Cacheable(cacheNames = "product", key = "#id")  
public Product getProductById(Long id) {  
    return productRepository.findById(id).orElseThrow();  
}
```

예시 흐름:

1. `getProductById(1L)` 호출 → 캐시에 없음 → DB 조회 → 결과를 캐시에 저장
2. 다시 `getProductById(1L)` 호출 → 캐시에 있음 → **DB 조회 없이 캐시에서 반환**
3. 캐시가 만료되거나 제거되면 → 다시 DB에서 조회

✅ 메모리에 저장된다는 것의 의미

- `caffeine`, `ehcache`, `redis` 등으로 저장 위치는 설정에 따라 달라질 수 있음
- `caffeine`은 JVM 메모리 내 로컬 캐시
- `redis`는 분산 환경에서 서버 간 공유되는 캐시

✅ 정리

질문

답변

한 번 호출되면 캐시되나?

✅ 예

다음 호출은 캐시에서 꺼내나?

✅ 예

DB 조회는 반복되지 않나?

✅ 캐시에 있으면 DB 접근 없음

캐시가 언제 없어지나?

❗ 설정한 TTL이나 최대 용량 초과 시 제거됨

처음 메서드가 호출될 때 결과를 캐시에 저장하고, 다음부터는 저장된 결과를 재사용하는 방식

✅ Caffeine 사용 용도

사용 사례	설명
메모리 내 캐시	DB 또는 API 호출 결과를 메모리에 임시 저장해 응답속도를 높임
Spring Cache 연동	<code>@Cacheable</code> 어노테이션 기반 캐시 구현 시, Caffeine을 캐시 구현체로 사용 가능
TTL(Time To Live), 최대 크기, LRU, LFU 등 다양한 캐시 제거 정책 제공	세밀한 캐시 관리 설정이 가능함
비동기 캐시 지원	<code>AsyncCache</code> 를 통해 비동기 로딩도 가능
Google Guava보다 빠름	Guava Cache보다 성능이 더 뛰어남 (후속 프로젝트로 시작 됨)

캐시정책

1. 캐싱 전략 (Caffeine Cache)

⚙️ properties

```
spring.cache.type=caffeine  
spring.cache.caffeine.spec=maximumSize=500,expireAfterWrite=600s
```

- Caffeine은 고성능 Java 캐시 라이브러리입니다
- `maximumSize=500`: 최대 500개의 항목을 캐시에 저장
- `expireAfterWrite=600s`: 캐시된 데이터는 600초(10분) 후에 만료
- 컨트롤러의 `@Cacheable` 어노테이션과 함께 작동하여 동일한 요청에 대한 반복적인 처리를 방지

동작 흐름 (Spring + Redis 기준)

1. 클라이언트가 `/persons` 요청
 2. `@Cacheable("persons")` 가 붙은 메서드 실행
 3. 캐시에 해당 **key**가 있는지 확인
 - 있으면 👉 캐시된 값 그대로 리턴 (DB 접근 X)
 - 없으면 👉 DB 조회 후, 결과를 캐시에 저장하고 리턴
 4. 이후 동일한 요청은 캐시에서 직접 가져옴
-


Redis를 사용하는 이유

- 다른 서버 간에도 공유 가능 → 분산 서버 환경에서 유리
- JVM이 재시작되더라도 캐시 유지 가능
- 대용량 캐시 운영에 더 안정적

상황 정리

java

 복사

 편집

```
@Cacheable("persons")
public List<Person> getAllPersons() {
    return personRepository.findAll();
}
```

- 처음 호출 시: DB에서 조회하고 → 결과를 "persons" 라는 키로 캐시에 저장
- 이후 호출 시: DB는 무시하고 캐시된 결과만 반환됨
- 문제: DB에 새로운 Person이 추가되거나 수정/삭제돼도, 기존 캐시는 그대로임 !

✅ 해결 방법: 캐시 무효화 (@CacheEvict)

DB를 변경하는 메서드에서 캐시를 지워주면 돼요.

java

📄 복사

✎ 편집

```
@CacheEvict(value = "persons", allEntries = true)
public void savePerson(Person person) {
    personRepository.save(person);
}
```

또는 삭제하는 경우에도:

java

📄 복사

✎ 편집

```
@CacheEvict(value = "persons", allEntries = true)
public void deletePerson(Long id) {
    personRepository.deleteById(id);
}
```

이렇게 하면 다음에 `getAllPersons()` 가 호출될 때 **DB에서 새로 조회**하고, 다시 캐시됩니다.

정리

변경 종류	캐시 영향	해결 방법
Person 추가/수정/삭제	캐시 무효화 안됨	<code>@CacheEvict</code> 로 캐시 삭제 필요
캐시 자동 갱신	❌ 기본은 안 됨	주기적 캐시 무효화 or 수동 무효화
TTL 설정	✅ 가능	Redis에서 설정 가능 (예: 10분 후 만료)

TTL 설정

@Bean

```
public RedisCacheManager cacheManager(RedisConnectionFactory connectionFactory) {  
    RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()  
        .entryTtl(Duration.ofMinutes(10)); // TTL 설정 (10분)  
  
    return RedisCacheManager.builder(connectionFactory)  
        .cacheDefaults(config)  
        .build();  
}
```

2) 비동기 처리:

@Async는 Spring에서 비동기 메서드 실행을 가능하게 해주는 어노테이션으로, 주로 시간이 오래 걸리는 작업을 별도 스레드에서 처리하고, 메인 스레드는 즉시 반환시키고 싶을 때 사용

언제 사용하나?

상황	설명
외부 API 호출	응답을 기다릴 필요 없이 바로 반환하고 나중에 처리
이메일 발송, 알림 전송	사용자는 기다리지 않아도 되고, 비동기로 처리
대용량 파일 업로드 후 처리	업로드 후 후속 처리를 백그라운드에서 수행
DB 작업 중 일부 통계/로그 기록	실시간 처리와 별개로 백그라운드로 처리 가능

✅ 어떤 이점이 있나?

장점	설명
🕒 응답 속도 향상	사용자는 느린 작업을 기다리지 않고 빠르게 응답받음
⚙️ 병렬 처리 가능	여러 작업을 동시에 실행시켜 효율적인 자원 사용 가능
🧵 멀티스레드 기반 처리	작업을 별도 스레드에서 실행해 메인 스레드와 분리됨

```
@Service
public class NotificationService {

    @Async
    public void sendEmail(String to, String message) {
        // 이메일 발송 로직 (시간 오래 걸리는 작업)
        System.out.println("이메일 전송 중: " + Thread.currentThread().getName());
    }
}
```

@Async의 리턴 타입: 반드시 void, Future, CompletableFuture 중 하나여야 함

비동기 환경 설정

2. 비동기 처리 설정

⚙️ properties

```
spring.task.execution.pool.core-size=10  
spring.task.execution.pool.max-size=20  
spring.task.execution.pool.queue-capacity=500
```

- `core-size=10`: 기본 스레드 풀 크기
- `max-size=20`: 최대 스레드 풀 크기
- `queue-capacity=500`: 대기 큐 크기
- `@Async` 어노테이션과 함께 사용되어 요청 처리를 비동기적으로 수행

3. 응답 압축 설정

properties

App

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

- GZIP 압축을 활성화하여 네트워크 대역폭 절약
- 1KB 이상의 응답에 대해서만 압축 적용
- 주요 MIME 타입에 대해 압축 적용

4. 톰캣 서버 최적화

properties

```
server.tomcat.max-threads=200  
server.tomcat.min-spare-threads=10  
server.tomcat.max-connections=10000  
server.tomcat.accept-count=100
```

- `max-threads=200`: 최대 동시 처리 스레드 수
- `min-spare-threads=10`: 유지할 최소 대기 스레드 수
- `max-connections=10000`: 최대 동시 연결 수
- `accept-count=100`: 대기 큐 크기

5. HTTP/2 및 캐시 컨트롤

⚙️ properties

```
server.http2.enabled=true
spring.mvc.cache.control.max-age=3600
spring.mvc.cache.control.no-cache=false
spring.mvc.cache.control.no-store=false
```

- HTTP/2 프로토콜 활성화로 성능 향상
- 브라우저 캐시 설정 (1시간 유효)
- 캐시 제어 헤더 설정

6. 스트리밍 응답 구현

📄 java

```
@GetMapping("/stream")
@ResponseBody
public ResponseEntity<StreamingResponseBody> streamData() {
    StreamingResponseBody stream = outputStream -> {
        for (int i = 0; i < 1000; i++) {
            outputStream.write(("Data " + i + "\n").getBytes());
            outputStream.flush();
        }
    };
    // ...
}
```

- 대용량 데이터를 청크 단위로 전송
- 메모리 효율적인 데이터 전송
- 실시간 데이터 스트리밍 지원

7. 의존성 추가

xml

```
<dependency>
  <groupId>com.github.ben-manes.caffeine</groupId>
  <artifactId>caffeine</artifactId>
  <version>3.1.8</version>
</dependency>
```

- Caffeine 캐시 라이브러리 추가
- Spring Boot 캐시 추상화와 통합

이러한 최적화들은 다음과 같은 이점을 제공합니다:

- 응답 시간 단축
- 서버 리소스 효율적 활용
- 네트워크 대역폭 절약
- 대용량 데이터 처리 효율성 향상
- 전반적인 시스템 성능 향상

3) 스트림 전송

- 데이터를 한 번에 모두 준비해서 보내는 방식이 아니라, 필요한 만큼 생성되는 대로 조금씩 클라이언트에 전송하는 방식.

✅ 언제 쓰면 좋을까?

상황	설명
📁 대용량 데이터 전송	CSV, 로그, 대형 텍스트 등 수 MB ~ GB 단위 데이터를 한꺼번에 로딩하지 않고 조각내어 전송
🕒 실시간 데이터 처리	서버에서 실시간으로 처리되는 데이터를 바로 클라이언트로 전달 (ex. 실시간 로그 보기)
🧠 메모리 절약	전체 데이터를 한 번에 메모리에 올리지 않아도 되어 효율적
🔄 비동기 처리	스레드가 오래 점유되지 않고, 클라이언트가 데이터를 받는 동안 계속 전송 가능

✅ 스트리밍과 일반 전송 차이

방식	설명	예시
일반 방식	모든 데이터를 한꺼번에 준비 → 한 번에 응답	<code>return List<Data></code>
스트리밍	데이터를 순차적으로 생성 → 부분 응답 즉시 전송	<code>StreamingResponseBody</code> , <code>SSE</code> , <code>WebFlux Flux</code> 등



요약

질문

답변

stream 전송은 데이터를 분할해서 보내는 방식인가요?

맞습니다!

언제 유용한가요?

대용량 데이터 전송, 실시간 처리, 메모리 절약 등

어떻게 작동하나요?

데이터를 생성하는 즉시 클라이언트에 전송 (`flush()` 중요)

ResponseEntity

java

📄 복사

✎ 편집

```
return ResponseEntity.ok()  
    .headers(headers)  
    .body(stream);
```

이 한 줄은 다음 세 가지를 의미합니다:

1. **HTTP 상태 코드:** 200 OK 설정
2. **응답 헤더 설정:** 파일 타입, 인코딩, 파일 이름 등 지정
3. **응답 바디:** 실시간 생성한 스트리밍 데이터 (`StreamingResponseBody`)

✓ 1. `ResponseEntity.ok()`

java

📄 복사

✎ 편집

```
ResponseEntity.ok()
```

- **HTTP 200 OK** 상태코드를 반환하는 `ResponseEntityBuilder` 를 생성
- 이후 `.headers()`, `.body()` 등 메서드를 체이닝 방식으로 이어서 사용할 수 있음

✓ 2. `.headers(headers)`

java

📄 복사

✎ 편집

```
.headers(headers)
```

- 사용자가 직접 설정한 `HttpHeaders` 객체를 응답에 포함시킴

예: 다운로드용 헤더 설정

java

📋 복사

✎ 편집

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.TEXT_PLAIN);
headers.setContentDisposition(
    ContentDisposition.attachment().filename("users.csv").build()
);
```

이렇게 하면 응답 헤더가 다음처럼 설정됩니다:

http

📋 복사

✎ 편집

```
Content-Type: text/plain
Content-Disposition: attachment; filename="users.csv"
```

→ 브라우저가 응답을 파일로 저장하도록 유도하는 핵심 역할

✓ 3. `.body(stream)`

java

📄 복사

✎ 편집

```
.body(stream);
```

- 실제로 전송할 응답 내용을 지정
- 여기서 `stream` 은 `StreamingResponseBody` 타입이므로, 서버는:
 - 데이터를 스트림 형식으로
 - 한 줄씩 점진적으로 생성해서
 - 클라이언트에 바로바로 전송하게 됩니다

✓ 전체 동작 흐름 요약

java

📄 복사

✎ 편집

```
return ResponseEntity.ok()  
    .headers(headers)  
    .body(stream);
```

구성 요소

설명

`ok()`

HTTP 상태코드 200 설정

`headers(...)`

파일 다운로드용 헤더 지정

`body(...)`

실시간으로 전송할 스트림 데이터 지정

✅ Spring에서 가능한 방식들

✅ 1. `StreamingResponseBody` + `JDBC cursor` 기반

java

📄 복사

✎ 편집

```
@GetMapping("/download")
public ResponseEntity<StreamingResponseBody> download() {
    StreamingResponseBody stream = outputStream -> {
        jdbcTemplate.query("SELECT * FROM big_table", rs -> {
            String line = rs.getString("name") + "," + rs.getInt("age") + "\n";
            outputStream.write(line.getBytes());
            outputStream.flush();
        });
    };

    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.TEXT_PLAIN);
    headers.setContentDisposition(ContentDisposition.attachment().filename("data.csv").build());

    return ResponseEntity.ok().headers(headers).body(stream);
}
```

✓ 2. JPA Stream<T> 사용 (주의사항 있음)

java

복사

편집

```
@Transactional
public Stream<User> streamAllUsers() {
    return userRepository.findAllBy(); // findAllBy() 리턴 타입이 Stream<User>
}
```

- JPA도 Stream<T>로 반환 가능
- 하지만 반드시 @Transactional 안에서 사용해야 함 → 세션을 열어둔 상태에서만 Stream 유효

✓ 3. WebFlux (Reactive 방식) - 대규모 실시간 처리에 적합

java

복사

편집

```
@GetMapping(value = "/flux", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<String> getFluxData() {
    return Flux.range(1, 100000)
        .map(i -> "data: " + i + "\n\n")
        .delayElements(Duration.ofMillis(10));
}
```

- 클라이언트에 Server-Sent Events로 실시간 전송 (text/event-stream)
- 대용량 데이터 처리에 최적화된 비동기 논블로킹 방식

✅ 핵심 차이 요약

항목	StreamingResponseBody	SSE (Server-Sent Events)
💡 목적	파일, 대량 데이터 스트리밍 (CSV, 로그 등)	실시간 이벤트/데이터 푸시 (주가, 알림 등)
📄 Content-Type	<code>application/octet-stream</code> , <code>text/plain</code> , etc.	<code>text/event-stream</code>
🔄 통신 방식	일반 HTTP Response (단방향)	HTTP 기반의 지속 연결된 이벤트 스트림 (단방향)
📦 메시지 구조	자유 형식 (직접 작성해야 함)	표준 이벤트 형식: <code>data:</code> , <code>event:</code> , <code>id:</code>
🧠 처리 방식	<code>StreamingResponseBody</code> , <code>OutputStream</code> 사용	<code>SseEmitter</code> , <code>Flux</code> , <code>EventSource</code> 사용
🌐 클라이언트 사용	<code>fetch()</code> , <code>XMLHttpRequest()</code> 등	<code>EventSource</code> API 전용
🕒 연결 유지	응답이 끝나면 종료	서버가 종료할 때까지 계속 열려 있음 (재연결 기능도 내장)

4) Tomcat server 설정 최적화

✓ 설정 항목 상세 설명

1. `server.tomcat.max-threads=200`

! 가장 중요한 설정 중 하나!

- Tomcat이 요청을 처리하기 위해 사용할 수 있는 최대 작업 스레드 수
- 이 숫자만큼 동시에 요청을 병렬 처리할 수 있음

항목	설명
기본값	200
예: 200 요청이 동시에 들어오면?	200개의 요청이 동시에 처리됨
그 이상 요청은?	대기열(<code>accept-count</code>)로 들어감

2. `server.tomcat.min-spare-threads=10`

- Tomcat이 항상 유지하려고 하는 최소 예비 스레드 수
- 트래픽이 증가할 때 빠르게 대응할 수 있도록 미리 준비된 스레드 수

항목	설명
기본값	10
의미	요청이 없어도 10개의 스레드는 대기 중
성능 영향	초기 요청 처리 속도 향상에 도움

3. `server.tomcat.max-connections=10000`

👤 클라이언트가 TCP 연결을 맺을 수 있는 최대 수

- Tomcat이 동시에 유지할 수 있는 최대 클라이언트 연결 수
- 요청 처리 스레드와는 별도로, 클라이언트의 커넥션 풀

항목	설명
기본값	8192 (Spring Boot 2.5 이후)
실제 처리 스레드 수와는 다름	연결만 유지할 수 있는 수 (keep-alive 포함)

요청이 완료되지 않아도 커넥션은 유지될 수 있음 (예: keep-alive)

4. `server.tomcat.accept-count=100`

- `max-threads` 스레드가 모두 바쁠 때 새 요청을 임시로 기다리게 하는 대기열 크기

항목	설명
기본값	100
의미	<code>max-threads</code> 가 모두 사용 증일 때, 최대 100개의 요청을 대기 가능
그 이상은?	커넥션 거부 또는 503 오류 발생 가능

요약 테이블

설정 키	의미	기본값	추천값 (트래픽에 따라)
max-threads	동시에 처리할 수 있는 최대 요청 수	200	200~1000
min-spare-threads	초기 예비 스레드 수	10	10~50
max-connections	동시에 연결 유지 가능한 클라이언트 수	8192	10000 이상 (고성능 시스템)
accept-count	대기 중인 요청 수 (큐)	100	200~500

✅ 예시: 요청 흐름

1. 클라이언트 100명의 요청이 동시에 올
 2. 0~199번 스레드까지 할당되어 즉시 처리 (최대 200)
 3. 201번째 요청은 accept-queue(100)에 대기
 4. 총 300개를 초과하는 요청이 오면 거절됨 (503 등)
-

✅ 성능 조정 팁

- CPU 코어 수 + 트래픽을 고려해서 `max-threads` 를 조절
- keep-alive 연결이 많을 경우 `max-connections` 도 늘리기
- 너무 높게 설정하면 오히려 리소스 소모 심해지니 부하 테스트 후 조정 필요

Tomcat 튜닝 체크리스트나 부하 테스트(k6, JMeter)

✓ 5. 성능 조정 예시

목표	설정 예시
동시에 최대 500 요청 처리	<code>max-threads: 500</code>
트래픽 대비 대기 큐 확보	<code>accept-count: 1000</code>
keep-alive 연결 많은 경우	<code>max-connections: 10000~20000</code>

✓ 요약

항목	설명
설정 파일	<code>server.tomcat.*</code> 설정으로 동시성 제어
컨트롤러	일부러 지연을 넣어서 부하 발생 테스트
테스트 도구	k6, JMeter 등 사용 가능
확인 포인트	처리 속도, 에러율, 응답시간, 서버 부하 확인

K6

k6는 개발자 친화적인 성능 테스트 오픈소스 도구로, JavaScript로 시나리오를 작성하여 HTTP API, 웹 애플리케이션의 부하 테스트를 수행할 수 있다.

■ TOTAL RESULTS

checks_total.....: 100 9.057269/s
checks_succeeded.....: 100.00% 100 out of 100
checks_failed.....: 0.00% 0 out of 100

✓ status is 200

HTTP

http_req_duration.....: avg=101.34ms min=100.57ms med=101.43ms max=103.75ms p(90)=101.71ms p(95)=101.81ms
{ expected_response:true }.....: avg=101.34ms min=100.57ms med=101.43ms max=103.75ms p(90)=101.71ms p(95)=101.81ms
http_req_failed.....: 0.00% 0 out of 100
http_reqs.....: 100 9.057269/s

EXECUTION

iteration_duration.....: avg=1.1s min=1.1s med=1.1s max=1.12s p(90)=1.1s p(95)=1.12s
iterations.....: 100 9.057269/s
vus.....: 10 min=10 max=10
vus_max.....: 10 min=10 max=10

NETWORK

data_received.....: 15 kB 1.4 kB/s
data_sent.....: 8.7 kB 788 B/s

running (11.0s), 00/10 VUs, 100 complete and 0 interrupted iterations
default ✓ [=====] 10 VUs 10s

✅ 테스트 구성 정보

항목	내용
VUs (가상 사용자 수)	100명
duration	10초 동안 테스트 수행
총 요청 수 (http_reqs)	1000건 (약 90건/초 처리)
실패율	0% (모든 요청 성공!)

✅ 결과 해석: 항목별 설명

◆ 체크 결과 (check 조건)

항목	설명
checks_total	check() 호출된 횟수: 1000번
✓ status is 200	응답 상태가 200인지 확인한 것
checks_succeeded	1000건 모두 성공 → 상태코드 200이 아닌 경우 없음
checks_failed	실패 0건, 매우 양호합니다 ✅

◆ HTTP 응답 시간

항목	설명
http_req_duration	요청 1건에 걸린 응답 시간
avg=102.24ms	평균 응답 시간 약 102ms
min=100.15ms / max=116.74ms	최소 100ms, 최대 116ms
p(90)=103.75ms	상위 90% 요청은 103ms 이하에서 응답됨
http_req_failed	실패 0건 (즉, 서버가 모두 잘 처리함)

✅ 결론: 서버 응답 속도 안정적이고 빠름!

✅ 최종 결과 분석

📦 HTTP 섹션

항목	설명
http_req_duration	응답시간 지표 (서버 처리 시간 포함)
avg=115.6ms	평균 응답 시간이 115.6 밀리초
p(90)=164.86ms	90%의 요청은 164ms 이하에서 응답
p(95)=224.64ms	95%는 225ms 이하
http_req_failed = 0.00%	✅ 실패 없음! 전부 200 OK 등의 정상 응답
http_reqs = 4500	총 HTTP 요청 수 (4500번 전송됨)

🧠 EXECUTION 섹션

항목	설명
iteration_duration	각 가상 사용자가 1회 요청 후 sleep까지 걸린 총 시간
평균: 1.13s	각 VU가 평균적으로 1.13초마다 반복
iterations: 4500	총 4500번 루프(요청) 수행됨
vus: 500	가상 유저 수 500명 유지
vus_max: 500	최대 500명까지 사용됨

🌐 NETWORK 섹션

항목	설명
data_received: 683kB	총 수신 데이터 양 (JSON 응답 등)
data_sent: 441kB	요청에 포함된 총 전송 데이터 양

약 66KB/s 수신, 42KB/s 송신 속도

◆ 실행 관련

항목	설명
iterations	총 1000회 요청 루프 실행
iteration_duration avg=1.1s	요청 + 슬립 포함 루프 시간이 약 1.1초
vus	100명 가상 유저가 동시에 활동 (최대/최소 동일)

◆ 네트워크 관련

항목	설명
data_received	152KB (응답 총량)
data_sent	87KB (요청 총량)

◆ 네트워크 관련

항목	설명
data_received	152KB (응답 총량)
data_sent	87KB (요청 총량)

● 최종 해석

"100명의 가상 유저가 10초 동안 서버에 1000건의 요청을 보냈고, 모두 성공, 평균 응답 속도는 약 **100ms**로 매우 우수한 결과입니다."

항목	목적	핵심 설정
Ramp-up	서버의 한계 구간 파악	<code>stages</code>
로그인 요청	인증 흐름 포함	<code>http.post() + token</code>
실패율 기준	테스트 품질 기준 적용	<code>thresholds</code>

JMeter

https://jmeter.apache.org/download_jmeter.cgi

1. apache-jmeter-5.6.3.zip
2. 압축해재 jmeter.bat 실행

주요 용어 정리

1. Throughput (처리량)

- 단위 시간당 처리된 요청 수
- JMeter에서는 **TPS (Transaction Per Second)** 단위로 표현됨

2. Response Time / Load Time (응답시간 / 로드시간)

- 요청을 보내고 응답이 완료되어 사용자 화면에 표시되기까지의 시간
- 시스템 성능 평가에 자주 사용

3. Latency (지연시간)

- 요청을 보낸 후 응답 수신이 시작되기까지의 시간

4. Think Time (사고 시간)

- 한 요청에 대한 응답을 받고 다음 요청을 보내기 전까지의 시간
- 사용자의 실제 사용 패턴을 시뮬레이션할 때 유용함

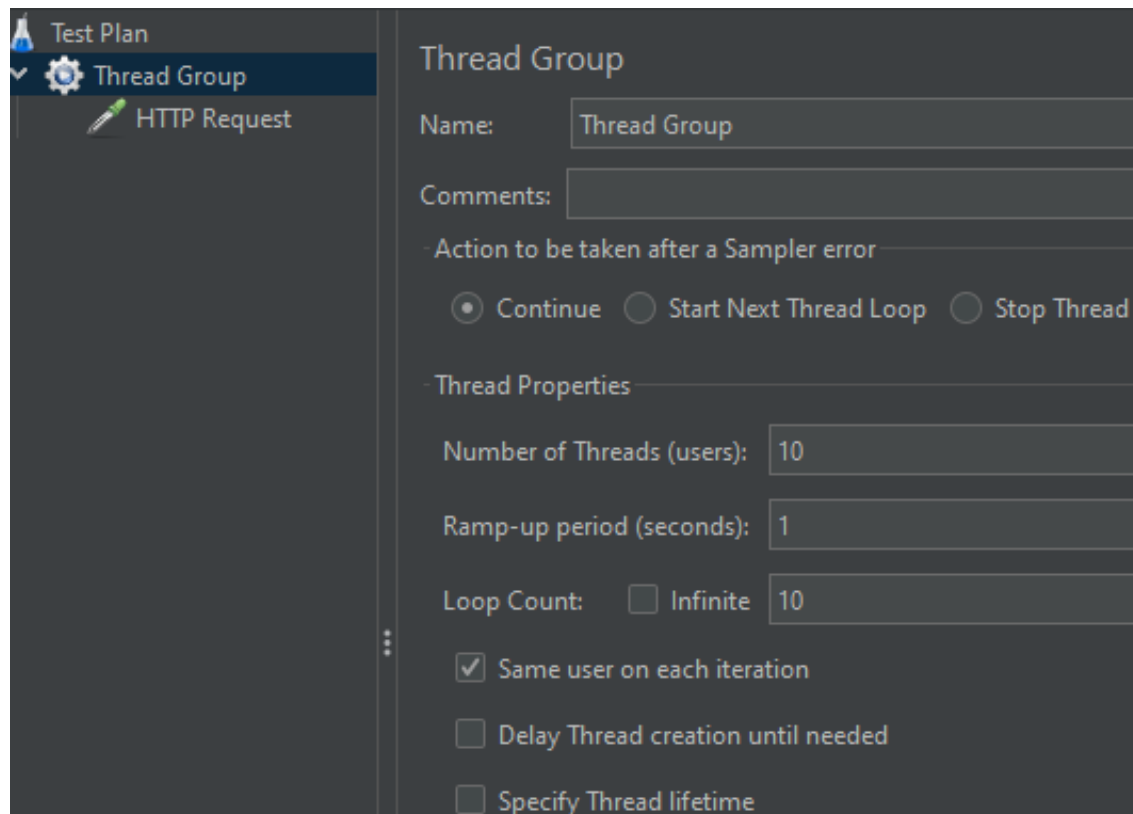
5. Request Interval Time (요청 간격 시간)

- 한 요청을 보내고 다음 요청을 보내기까지의 전체 시간
- `Request Interval Time = Load Time + Think Time`

6. Load Time vs Latency

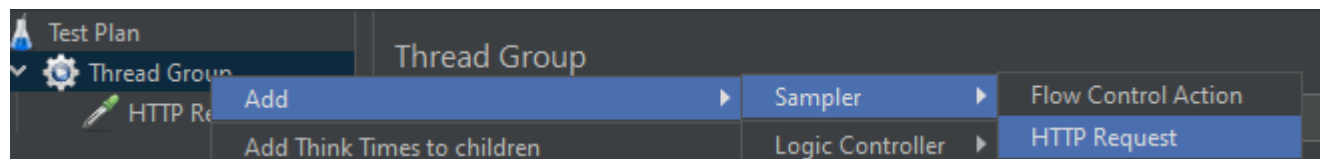
- 아래 그림에서 두 시간의 관계를 시각화

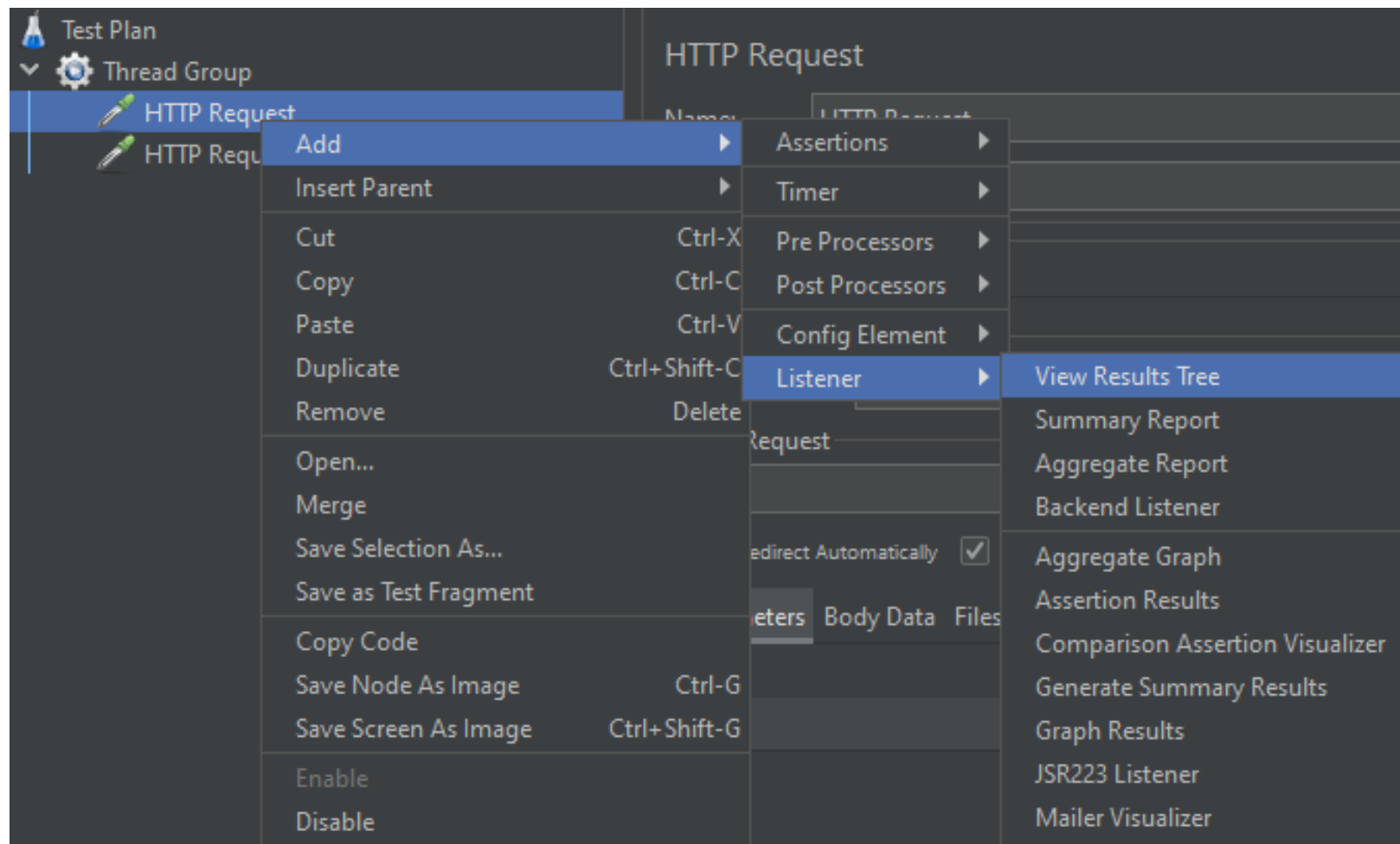
구간	설명
Latency	서버가 응답을 시작할 때까지의 대기 시간
Load Time	전체 응답이 완료될 때까지의 시간
Think Time	응답을 받고 다음 요청을 보내기 전까지의 시간
Request Interval	하나의 요청 사이 전체 간격 (Load + Think)

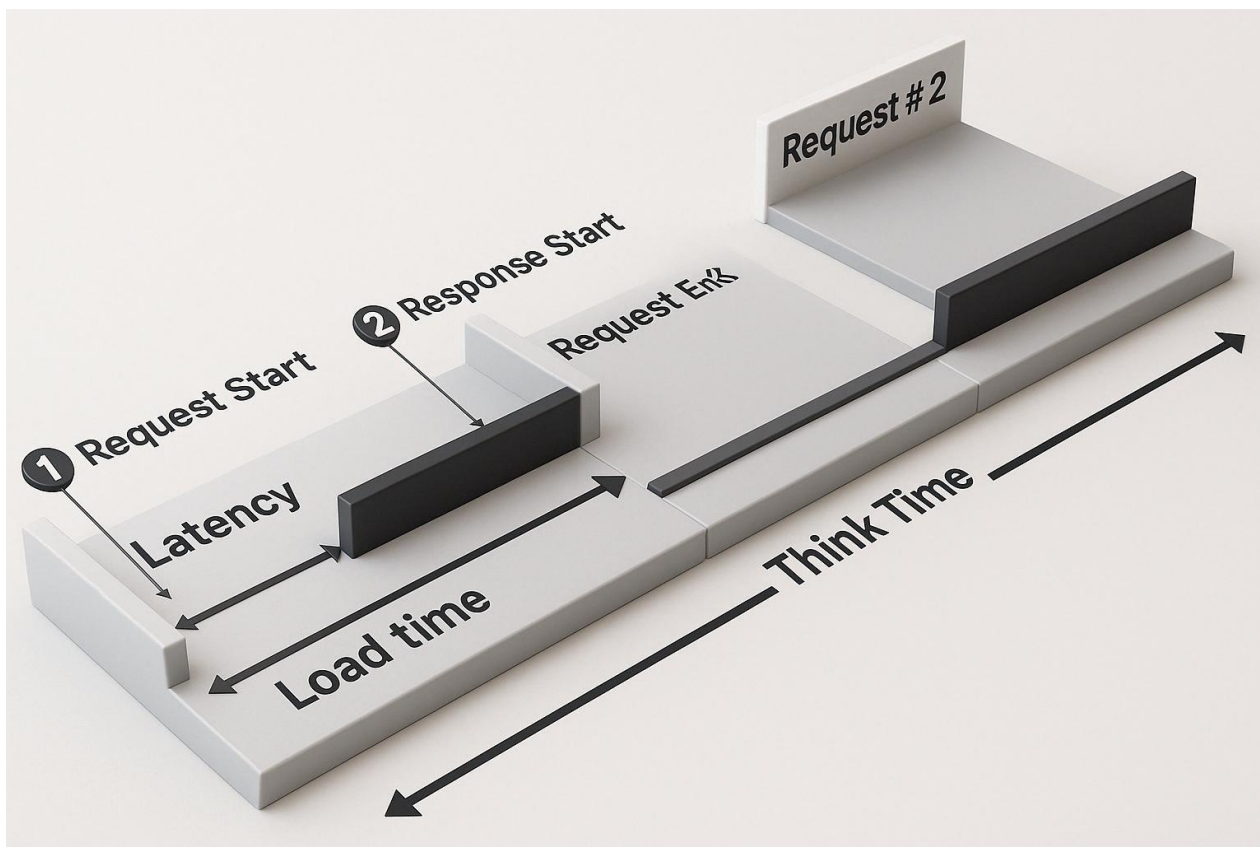


동시접속자수

각각 10번씩 접속







Test Plan

- Thread Group
 - HTTP Request
 - View Results Tree
 - HTTP Request

View Results Tree

Name: View Results Tree

Comments:

- Write results to file / Read from file

Filename

Search: ☐ Case sensitive ☐ Regular exp.

Text

- ✓ HTTP Request
- ✓ HTTP Request
- ✓ HTTP Request

Sampler result Request Response data

Thread Name: Thread Group 1-1
Sample Start: 2025-04-19 22:44:58 KST
Load time: 106
Connect Time: 2
Latency: 106
Size in bytes: 200
Sent bytes: 120
Headers size in bytes: 162
Body size in bytes: 38
Sample Count: 1
Error Count: 0
Data type ("text"|"bin"|""): text
Response code: 200
Response message:

HTTPSampleResult fields:
ContentType: text/plain; charset=UTF-8
DataEncoding: UTF-8

단위: 밀리세컨드 106 은 0.1초

Test Plan

Thread Group

HTTP Request

View Results Tree

Summary Report

HTTP Request

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Browse...

Log/Display Only:

Errors

Successes

Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	6	103	102	108	2.11	0.00%	58.8/min	0.19	0.11	199.5
TOTAL	6	103	102	108	2.11	0.00%	58.8/min	0.19	0.11	199.5

Min: 가장 빠른 access

Max: 가장 오래걸린 access

Average: 평균 access 시간

Throughput: 초당 access 횟수

Error: 에러

표준편차: access 편차

Spring Actuator

- **Spring Boot** 애플리케이션의 **운영 및 모니터링 기능**을 손쉽게 제공하는 모듈입니다. 시스템의 상태를 확인하고, 메트릭, 트래픽, 로그, 환경 변수 등의 정보를 HTTP 엔드포인트로 제공

Spring Boot Actuator란?

Spring Actuator는 다음과 같은 운영(운용) 관련 기능을 자동으로 구성해줍니다:

기능 구분	설명
 헬스 체크	시스템이 살아 있는지 (<code>/actuator/health</code>)
 메트릭	CPU 사용률, 메모리, GC, HTTP 요청 수 등 (<code>/actuator/metrics</code>)
 환경 정보	환경 변수, 시스템 속성, 프로파일 정보 (<code>/actuator/env</code> , <code>/actuator/configprops</code>)
 빈 정보	등록된 스프링 빈 목록 (<code>/actuator/beans</code>)
 매핑 정보	어떤 URL이 어떤 컨트롤러에 매핑되어 있는지 (<code>/actuator/mappings</code>)
 로그 레벨 관리	런타임에 로그 레벨 동적 변경 (<code>/actuator/loggers</code>)
 보안	민감한 정보는 기본적으로 보호되며, Spring Security와 연계 가능



주요 엔드포인트 예시

엔드포인트

설명

/actuator/health

서비스가 정상인지 (200 OK, DOWN 등)

/actuator/info

앱의 커스텀 정보 (버전, 작성자 등)

/actuator/metrics

메트릭 키 목록 및 상세

/actuator/metrics/jvm.memory.used

JVM 메모리 사용량

/actuator/mappings

어떤 요청이 어떤 컨트롤러에 매핑되어 있는지

/actuator/env

환경 변수 및 프로퍼티 확인

/actuator/loggers

로그 레벨 실시간 조절



예시: Health 상태 확인

bash

📄 복사

✎ 편집

```
curl http://localhost:8080/actuator/health
```


응답 예:

json

📄 복사


✎ 편집

```
{  
  "status": "UP"  
}
```

 pom.xml +4 ✓

```
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
</dependencies>

<build>
```

 application.properties +12 ✓

Actuator 설정

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
management.endpoint.metrics.enabled=true
management.endpoint.prometheus.enabled=true
management.metrics.export.prometheus.enabled=true
management.endpoint.loggers.enabled=true
management.endpoint.env.enabled=true
management.endpoint.beans.enabled=true
management.endpoint.threaddump.enabled=true
management.endpoint.heapdump.enabled=true
```

1. /actuator/health - 애플리케이션의 상태 정보
2. /actuator/metrics - 다양한 메트릭 정보
3. /actuator/prometheus - Prometheus 형식의 메트릭
4. /actuator/loggers - 로깅 설정 관리
5. /actuator/env - 환경 변수 정보
6. /actuator/beans - Spring Bean 정보
7. /actuator/threaddump - 스레드 덤프
8. /actuator/heapdump - 힙 덤프

애플리케이션을 실행하면 기본적으로 /actuator 경로로 접근할 수 있습니다.
모든 엔드포인트는 /actuator/{endpoint} 형식으로 접근 가능

1. 특정 엔드포인트만 노출:

⚙️ properties

```
management.endpoints.web.exposure.include=health,metrics,info
```

2. Actuator 엔드포인트에 대한 보안 설정:

⚙️ properties

```
management.endpoints.web.base-path=/manage  
management.endpoints.web.exposure.include=*  
management.endpoints.web.cors.allowed-origins=http://example.com  
management.endpoints.web.cors.allowed-methods=GET,POST
```

http://localhost:8081/actuator/health

```
{"status":"UP","components":{"db":{"status":"UP","details":{"database":"H2","validationQuery":"isValid()"}}, "diskSpace":{"status":"UP","details":{"total":475914039296,"free":276704219136,"threshold":10485760,"path":"D:\\autoever_hyundai\\source1\\05.actuator\\.","exists":true}}, "ping":{"status":"UP"}, "ssl":{"status":"UP","details":{"validChains":[],"invalidChains":[]}}}}
```

컴포넌트	상태	설명
전체	UP	애플리케이션 전체적으로 정상
db	UP	H2 DB 연결 이상 없음
diskSpace	UP	디스크 공간 충분함 (276GB)
ping	UP	기본 응답 체크 성공
ssl	UP	SSL 설정은 정상이나 체인은 없음

<http://localhost:8081/actuator/metrics>

```
{"names":["application.ready.time","application.started.time","disk.free","disk.total","executor.active","executor.completed","executor.pool.core","executor.pool.max","executor.pool.size","executor.queue.remaining","executor.queued","hikaricp.connections","hikaricp.connections.acquire","hikaricp.connections.active","hikaricp.connections.creation","hikaricp.connections.idle","hikaricp.connections.max","hikaricp.connections.min","hikaricp.connections.pending","hikaricp.connections.timeout","hikaricp.connections.usage","http.server.requests","http.server.requests.active","jdbc.connections.active","jdbc.connections.idle","jdbc.connections.max","jdbc.connections.min","jvm.buffer.count","jvm.buffer.memory.used","jvm.buffer.total.capacity","jvm.classes.loaded","jvm.classes.unloaded","jvm.compilation.time","jvm.gc.live.data.size","jvm.gc.max.data.size","jvm.gc.memory.allocated","jvm.gc.memory.promoted","jvm.gc.overhead","jvm.gc.pause","jvm.info","jvm.memory.committed","jvm.memory.max","jvm.memory.usage.after.gc","jvm.memory.used","jvm.threads.daemon","jvm.threads.live","jvm.threads.peak","jvm.threads.started","jvm.threads.states","logback.events","process.cpu.time","process.cpu.usage","process.start.time","process.uptime","system.cpu.count","system.cpu.usage","tomcat.sessions.active.current","tomcat.sessions.active.max","tomcat.sessions.alive.max","tomcat.sessions.created","tomcat.sessions.expired","tomcat.sessions.rejected"]}]}
```

✅ 애플리케이션 상태 관련

메트릭 이름	설명
<code>application.started.time</code>	애플리케이션이 시작되기까지 걸린 시간 (ms)
<code>application.ready.time</code>	애플리케이션이 "ready" 상태가 되기까지 걸린 시간 (ms)

🔧 스레드풀 (Executor) 관련

메트릭 이름	설명
<code>executor.active</code>	현재 실행 중인 스레드 수
<code>executor.completed</code>	완료된 작업 수
<code>executor.pool.core</code>	core pool size
<code>executor.pool.max</code>	max pool size
<code>executor.pool.size</code>	현재 pool 크기
<code>executor.queue.remaining</code>	남은 큐 공간
<code>executor.queued</code>	큐에 대기 중인 작업 수

💾 디스크 관련

메트릭 이름	설명
<code>disk.free</code>	사용 가능한 디스크 공간 (bytes)
<code>disk.total</code>	전체 디스크 용량 (bytes)

HikariCP (DB 커넥션 풀) 관련

메트릭 이름	설명
<code>hikaricp.connections</code>	전체 커넥션 수
<code>hikaricp.connections.active</code>	현재 사용 중인 커넥션 수
<code>hikaricp.connections.idle</code>	사용 가능 커넥션 수
<code>hikaricp.connections.pending</code>	커넥션 요청 대기 수
<code>hikaricp.connections.max</code>	설정된 최대 커넥션 수
<code>hikaricp.connections.min</code>	설정된 최소 커넥션 수
<code>hikaricp.connections.timeout</code>	커넥션 획득 시도 후 타임아웃 수
<code>hikaricp.connections.usage</code>	커넥션 평균 사용 시간
<code>hikaricp.connections.acquire</code>	커넥션 획득 평균 시간
<code>hikaricp.connections.creation</code>	커넥션 생성 평균 시간

HTTP 요청 관련

메트릭 이름	설명
<code>http.server.requests</code>	HTTP 요청에 대한 총 응답 정보 (타이밍, 수 등)
<code>http.server.requests.active</code>	현재 처리 중인 요청 수

JVM 관련

메트릭 이름	설명
<code>jvm.memory.used</code>	JVM 메모리 사용량
<code>jvm.memory.committed</code>	JVM에서 커밋된 메모리
<code>jvm.memory.max</code>	JVM 메모리 최대값
<code>jvm.threads.live</code>	현재 살아있는 스레드 수
<code>jvm.threads.daemon</code>	데몬 스레드 수
<code>jvm.threads.peak</code>	피크 스레드 수
<code>jvm.threads.started</code>	시작된 전체 스레드 수
<code>jvm.classes.loaded</code>	로드된 클래스 수
<code>jvm.classes.unloaded</code>	언로드된 클래스 수
<code>jvm.gc.pause</code>	GC 일시중지 시간
<code>jvm.gc.memory.allocated</code>	할당된 메모리량
<code>jvm.gc.memory.promoted</code>	영역 간 이동된 메모리량
<code>jvm.gc.live.data.size</code>	GC 후 살아남은 데이터 크기
<code>jvm.info</code>	JVM 버전, 벤더 등 정보

시스템/프로세스 관련

메트릭 이름	설명
<code>process.cpu.time</code>	현재 JVM 프로세스의 누적 CPU 사용 시간
<code>process.cpu.usage</code>	JVM CPU 사용률 (0~1)
<code>process.start.time</code>	프로세스 시작 시간 (timestamp)
<code>process.uptime</code>	프로세스 시작 이후 경과 시간 (초)
<code>system.cpu.count</code>	CPU 코어 수
<code>system.cpu.usage</code>	전체 시스템의 CPU 사용률 (0~1)

Tomcat 세션 관련

메트릭 이름	설명
<code>tomcat.sessions.active.current</code>	현재 활성 세션 수
<code>tomcat.sessions.active.max</code>	최대 활성 세션 수
<code>tomcat.sessions.created</code>	생성된 총 세션 수
<code>tomcat.sessions.expired</code>	만료된 세션 수
<code>tomcat.sessions.rejected</code>	거부된 세션 수

로깅

메트릭 이름	설명
<code>logback.events</code>	로그 이벤트 수 (레벨별 count 가능)

사용 예

메트릭의 값을 보려면 아래처럼 요청합니다:

bash

 복사

 편집

```
curl http://localhost:8081/actuator/metrics/jvm.memory.used
```

결과:

json

 복사

 편집

```
{
  "name": "jvm.memory.used",
  "measurements": [{"statistic": "VALUE", "value": 12345678}],
  "availableTags": [{"tag": "area", "values": ["heap", "nonheap"]}]}
}
```


Prometheus와 Grafana 대시보드

- <https://prometheus.io/download/>
- <https://grafana.com/grafana/download?platform=windows>
- 설치

1. 먼저 Prometheus 의존성을 추가하겠습니다:

 pom.xml +4 •

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-registry-prometheus</artifactId>
    </dependency>
</dependencies>

<build>
```

1.Prometheus 설정:

- Prometheus를 다운로드하고 설치
(<https://prometheus.io/download/>)
- prometheus.yml 파일을 다음과 같이 설정:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'spring-actuator'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:8081']  # Spring Boot 애플리케이션의 포트
```

2. 실행

```
cd C:\prometheus-2.52.0-windows-amd64
prometheus.exe --config.file=prometheus.yml
http://localhost:9090
```

✅ 1단계: Prometheus 바이너리 다운로드

▶ 공식 다운로드 페이지에서 OS에 맞는 버전 선택

- 예: Windows의 경우 `prometheus-*-windows-amd64.zip`
- 예: macOS의 경우 `prometheus-*-darwin-amd64.tar.gz`

▶ 다운로드 후 압축 해제

예) `C:\prometheus-2.52.0-windows-amd64\`

✅ 2단계: `prometheus.yml` 설정 파일 수정

압축 해제한 폴더 안에 있는 `prometheus.yml` 을 편집:

```
yaml

global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'spring-actuator'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['localhost:8081']
```

✅ 3단계: Prometheus 실행

▶ Windows

- `cmd` 또는 PowerShell 실행
- Prometheus 폴더로 이동:

```
cmd

cd C:\prometheus-2.52.0-windows-amd64
```

- 실행:

```
cmd

prometheus.exe --config.file=prometheus.yml
```

▶ macOS / Linux

```
bash

cd ~/prometheus-2.52.0-darwin-amd64
./prometheus --config.file=prometheus.yml
```

✅ 4단계: 브라우저로 확인

접속: <http://localhost:9090>

+ 확인 방법

- 검색창에: `application_ready_time_seconds` 같은 메트릭 이름 입력
- `Graph` 탭에서 시각화 가능

2. Grafana 설정:

- Grafana를 다운로드하고 설치합니다 (<https://grafana.com/grafana/download>)
- Grafana 웹 인터페이스에 접속 (기본 포트: 3000)
- 데이터 소스로 Prometheus를 추가
- 대시보드 생성 또는 기존 대시보드 import

3. 추천 대시보드:

- Spring Boot 2.1 Statistics (ID: 10280)
- JVM (Micrometer) (ID: 4701)
- Spring Boot Dashboard (ID: 6756)

Localhost:3000

Id: admin

Pass: admin

▶ 1단계: Grafana 설치

공식 [다운로드 페이지](#) 에서 OS에 맞는 설치 파일 다운로드

- Windows: `.msi` 설치 파일
- macOS: `brew install grafana`
- Linux (예: Ubuntu):


```
bash

sudo apt-get install -y adduser libfontconfig1
wget https://dl.grafana.com/oss/release/grafana_10.2.3_amd64.deb
sudo dpkg -i grafana_10.2.3_amd64.deb
```

✅ Grafana 웹 접속

- 브라우저에서: <http://localhost:3000>
- 기본 로그인 정보:
 - 사용자: `admin`
 - 비밀번호: `admin` (처음 로그인 시 변경 요구)

✅ Prometheus 데이터 소스 연결


1. 왼쪽 사이드바 →  **Configuration** → **Data Sources**
2. “Add data source” 클릭
3. **Prometheus** 선택
4. URL에 Prometheus 주소 입력:

```
arduino

http://localhost:9090
```


5. “Save & Test” 클릭 → 성공 메시지 확인

✅ 대시보드 만들기


1. 왼쪽  메뉴 → **Dashboard** → **New Dashboard**
2. “Add new panel” 클릭
3. PromQL 입력 예시:

```
prometheus

http_server_requests_seconds_count
```

4. 시각화 형태(Line, Bar 등) 선택
5. 저장: 우상단  Save

✅ 추천 Grafana 대시보드 가져오기 (Import)

1. 왼쪽  메뉴 → Import
2. 아래 ID 입력 (공식 공유 대시보드):
 - **Spring Boot / Micrometer Metrics:** 4701
3. Prometheus 데이터 소스 선택
4. "Import" 클릭

Prometheus vs Grafana 비교

항목	Prometheus 자체 UI	Grafana
 메트릭 탐색	○ (Query 실행 가능)	○ (더 직관적이고 다양한 편집 도구 제공)
 시각화 스타일	기본 Line 그래프	Line, Bar, Gauge, Heatmap, Table 등 다양
 대시보드 저장	X (일시적인 그래프)	○ (대시보드 구성 및 저장, 공유 가능)
 필터/조건 검색	제한적	변수(Variables), 조건 필터, 템플릿 지원
 알람 설정	Alertmanager 필요	내장 알람 기능 + Slack, 이메일 등 통합
 디자인/UX	단순	커스터마이징 가능 (색상, 크기, 배치 등)
 여러 소스 통합	Prometheus만	Prometheus 외에도 MySQL, Elasticsearch, Loki 등 수십 개 데이터 소스 통합 가능

Grafana

Home

Bookmarks

Starred

Dashboards

Playlists

Snapshots

Library panels

Shared dashboards

Explore

Drilldown

Alerting

Connections

Add new connection

Data sources

Administration

Home > Connections > Data sources

Search or jump to...

ctrl+k

+ v

?

📄

Data sources

View and manage your connected data source connections

Search by name or type

Sort by A-Z

prometheus

Prometheus | http://localhost:8081 | default

Build a dashboard

Explore

prometheus-1

Prometheus | http://localhost:9090

Build a dashboard

Explore

prometheus-2

Prometheus

Build a dashboard

Explore

클릭

Outline

prometheus-1

Split

Add

Last 1 hour

Run query

Queries

Graph

Raw Prometheus

Kick start your query

Metrics browser

rate(http_server_requests_seconds_count[1h])

Options

Legend: Auto

Format: Time series

Step: auto

Type: Both

Exemplars: false

Add query

Query inspector

Graph

Lines

Bars

Points

Stacked lines

Stacked bars

Time	Rate
10:05	0.0000
10:10	0.00028
10:50	0.00055
11:00	0.00055

{app="prometheus", error="none", exception="none", instance="localhost:8081", job="prometheus", method="GET", outcome="CLIENT_ERROR", status="404", uri="/"} (green)

{app="prometheus", error="none", exception="none", instance="localhost:8081", job="prometheus", method="GET", outcome="SUCCESS", status="200", uri="/"} (yellow)

{app="prometheus", error="none", exception="none", instance="localhost:8081", job="prometheus", method="GET", outcome="SUCCESS", status="200", uri="/actuator/metrics"} (blue)

{app="prometheus", error="none", exception="none", instance="localhost:8081", job="prometheus", method="GET", outcome="SUCCESS", status="200", uri="/actuator/prometheus"} (orange)

{app="prometheus", error="none", exception="none", instance="localhost:8081", job="prometheus", method="GET", outcome="SUCCESS", status="200", uri="/api/persons"} (red)