와플스튜디오 Spring Seminar

세미나장: 정원식

2023.09.20.(수) 19:00

Week1

Table of Contents

- Week0 과제 리뷰
- 스프링 기본 동작 원리
  - ComponentScan
  - RequestHandlerMapping
  - DispatcherServlet
- 데이터베이스
  - 영속성
  - RDB
  - JPA
- Week1 과제 관련 공지

Week0 과제 리뷰

- getAccessToken 함수
- ExceptionHandler 글로벌 적용
- 패키지 구조
- 인자 이름 명시
- When
- 중괄호

스프링 동작 원리

Run SeminarApplication

- 빈으로 등록할 대상들을 스캔
- 빈 등록
- HTTP 요청 – 컨트롤러 맵핑

- 8080 포트 열고 HTTP 요청을 받기 시작

- HTTP 요청이 들어오면 맵핑된 컨트롤러를 찾아 응답 처리

스프링 동작 원리

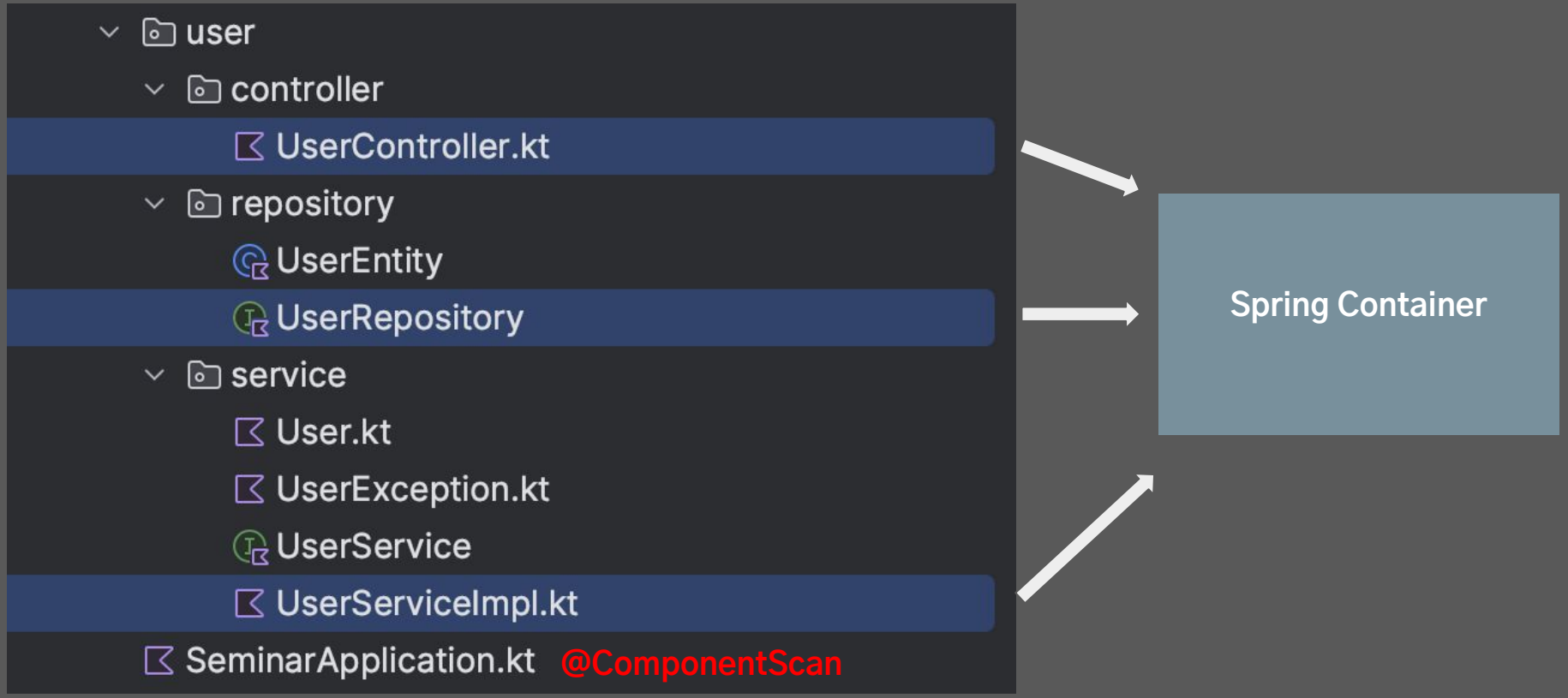
빈 등록 – 컴포넌트 스캔

- ComponentScan: 빈으로 등록할 대상들을 찾는 과정
- @ComponentScan을 가진 클래스의 하위 경로를 대상
- @SpringBootApplication은 @ComponentScan을 포함한 어노테이션
- SeminarApplication.kt를 기준으로 빈 등록 대상을 찾는다(com.wafflestudio.seminar.spring2023)

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

스프링 동작 원리

빈 등록


user
controller
UserController.kt
repository
UserEntity
UserRepository
service
User.kt
UserException.kt
UserService
UserServiceImpl.kt
SeminarApplication.kt
@ComponentScan
Spring Container

스프링 동작 원리

HTTP 요청과 RestController의 함수를 맵핑

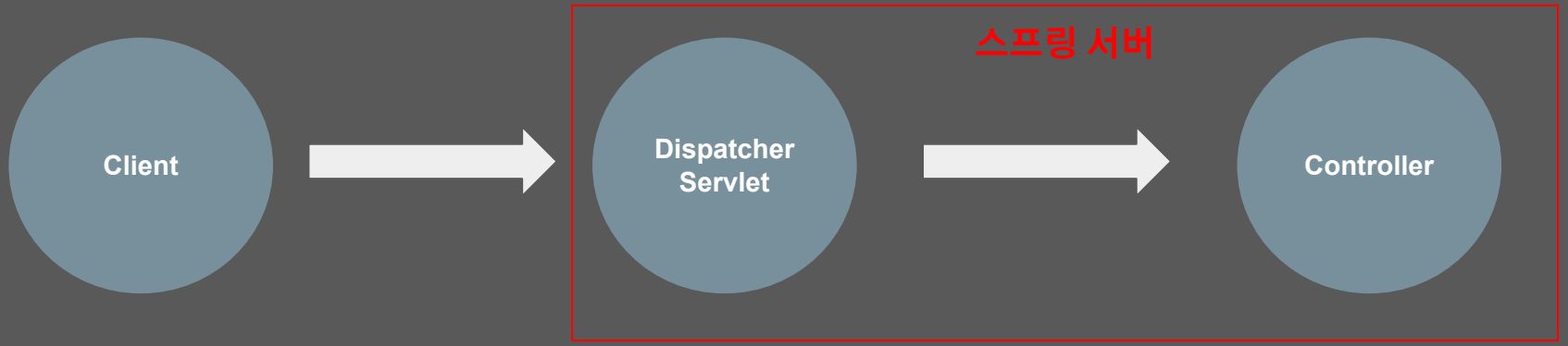
Spring Container

RequestHandlerMapping

HTTP Request	Handler Method
GET /api/v1/users/me	UserController.me
GET /api/v1/playlist-groups	PlaylistController.groups

스프링 동작 원리

DispatcherServlet, HTTP 요청 처리



HTTP 요청

RequestHandlerMapping을

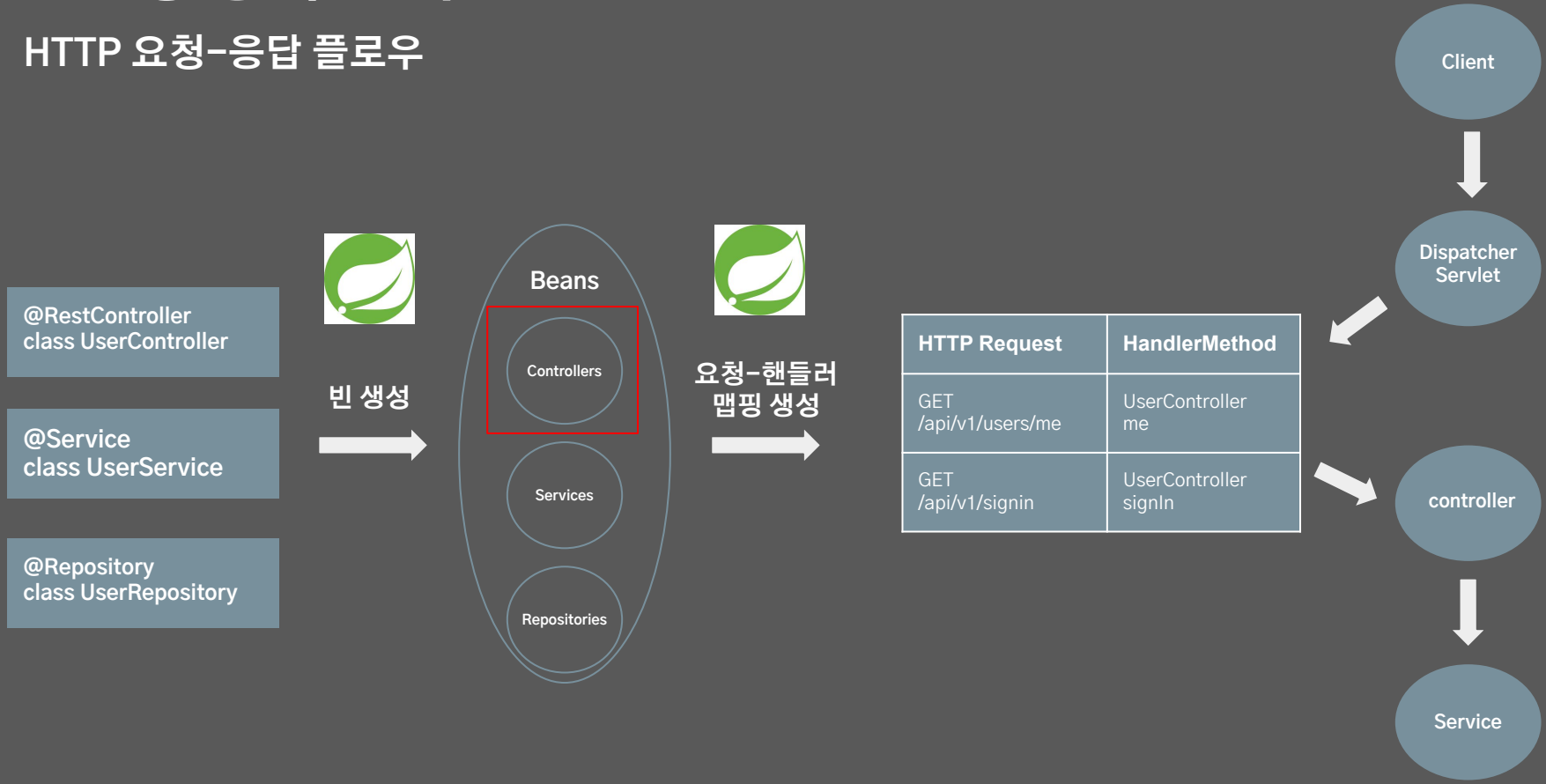
참고하여 요청을 처리할 Controller

호출

요청 처리

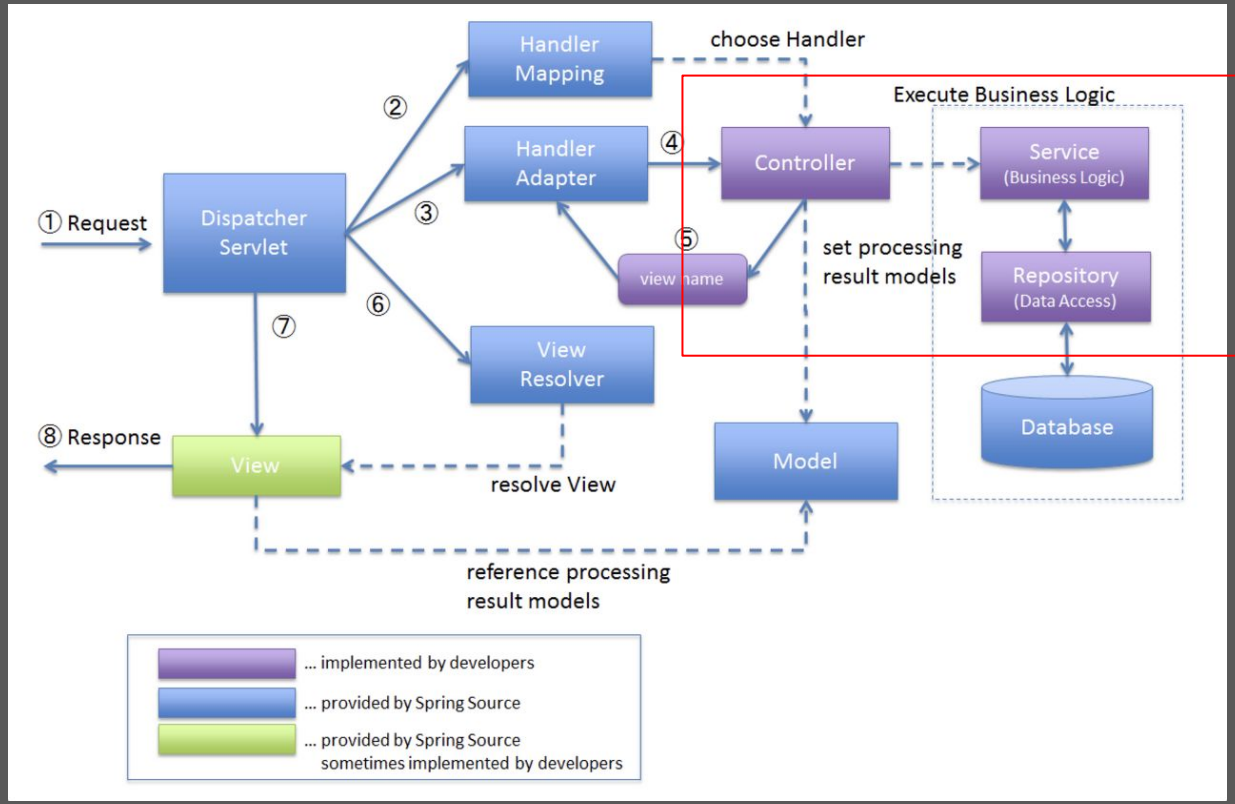
스프링 동작 원리

HTTP 요청-응답 플로우



스프링 동작 원리

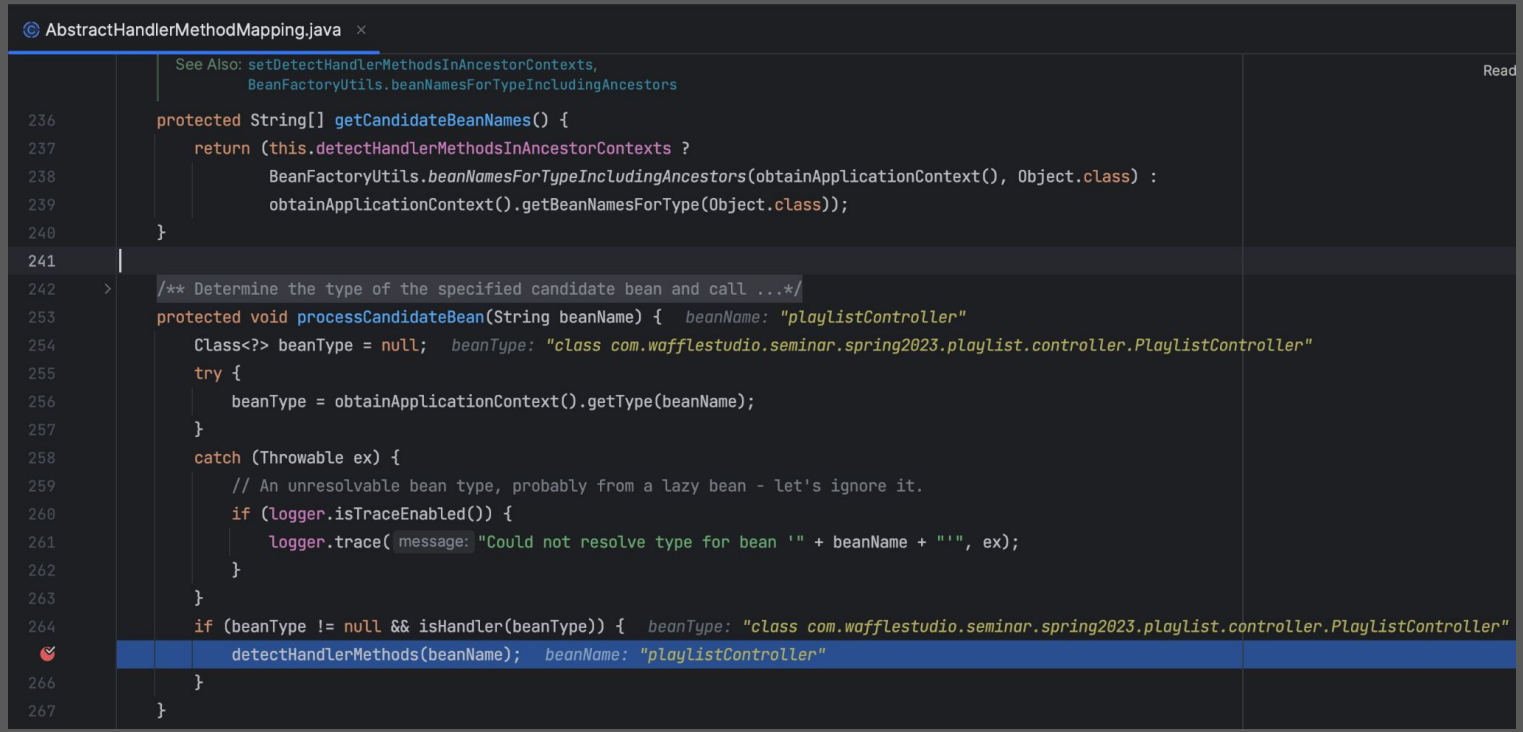
스프링의 강력한 기능을 통해 비지니스 로직에만 집중



우리가 직접 구현하는 영역

스프링 동작 원리

BreakPoint 1. HandlerMethodMapping



스프링 동작 원리

BreakPoint 2. DispatcherServlet

DispatcherServlet.java

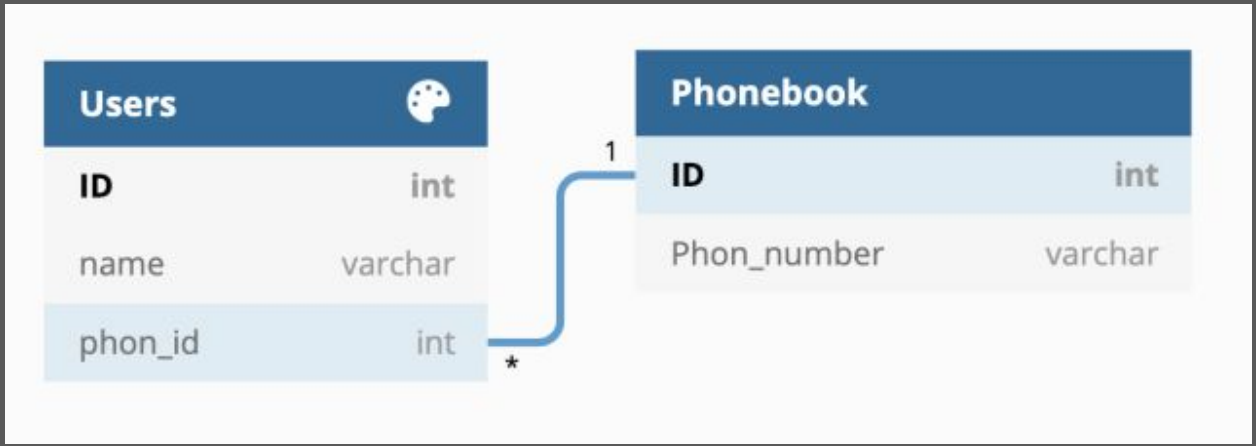
```
1061            }
1062
1063            // Determine handler adapter for the current request.
1064            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
1065
1066            // Process last-modified header, if supported by the handler.
1067            String method = request.getMethod();
1068            boolean isGet = HttpMethod.GET.matches(method);
1069            if (isGet || HttpMethod.HEAD.matches(method)) {
1070                long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
1071                if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {
1072                    return;
1073                }
1074            }
1075
1076            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
1077                return;
1078            }
1079
1080            // Actually invoke the handler.
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
1082
1083            if (asyncManager.isConcurrentHandlingStarted()) {
1084                return;
1085            }
1086
1087            applyDefaultViewName(processedRequest, mv);
1088            mappedHandler.applyPostHandle(processedRequest, response, mv);
1089        }
```

영속성 (Persistence)

- 영속성이란 프로그램이 종료되어도 데이터가 유지되는 성질
- 온전한 서비스를 위해서는 데이터의 영속성이 필수적
- 파일 시스템(txt, csv, xlsx), 별도의 데이터베이스 서버(MySQL, MongoDB)

관계형 데이터베이스 (Relational Database)

- 데이터 항목들은 행(row)에 저장되고, 항목의 속성은 열(column)이라고 표현
- 테이블의 행과 행이 연결되는 관계를 맺을 수 있음
- 테이블 간의 관계는 일 대 일(1:1), 일 대 다(1:N), 다 대 다(N:N) 의 관계



관계형 데이터베이스 (Relational Database)

테이블 간 관계(1:1, 1:N, M:N)

1:N 관계

한 아티스트가 여러 개의 앨범을 가질 수 있다.

albums	
PK	id bigint
	title varchar(100) not null
	image varchar(200) not null
	artist_id bigint not null

artists	
PK	id bigint not null
UK	name varchar(30)

M:N 관계

한 아티스트가 여러 개의 곡을 가질 수 있다.
반대로 한 곡이 여러 명의 아티스를 가질 수 있다.

songs	
PK	id bigint
	title varchar(255) not null
	duration int not null
	album_id bigint not null

song_artists	
PK	id bigint not null
	artist_id bigint not null
	song_id bigint not null

artists	
PK	id bigint not null
UK	name varchar(255)

관계형 데이터베이스 (Relational Database)

SQL (Structured Query Language)

- RDB에서 사용하는 언어
- 데이터를 검색, 추가, 수정, 제거

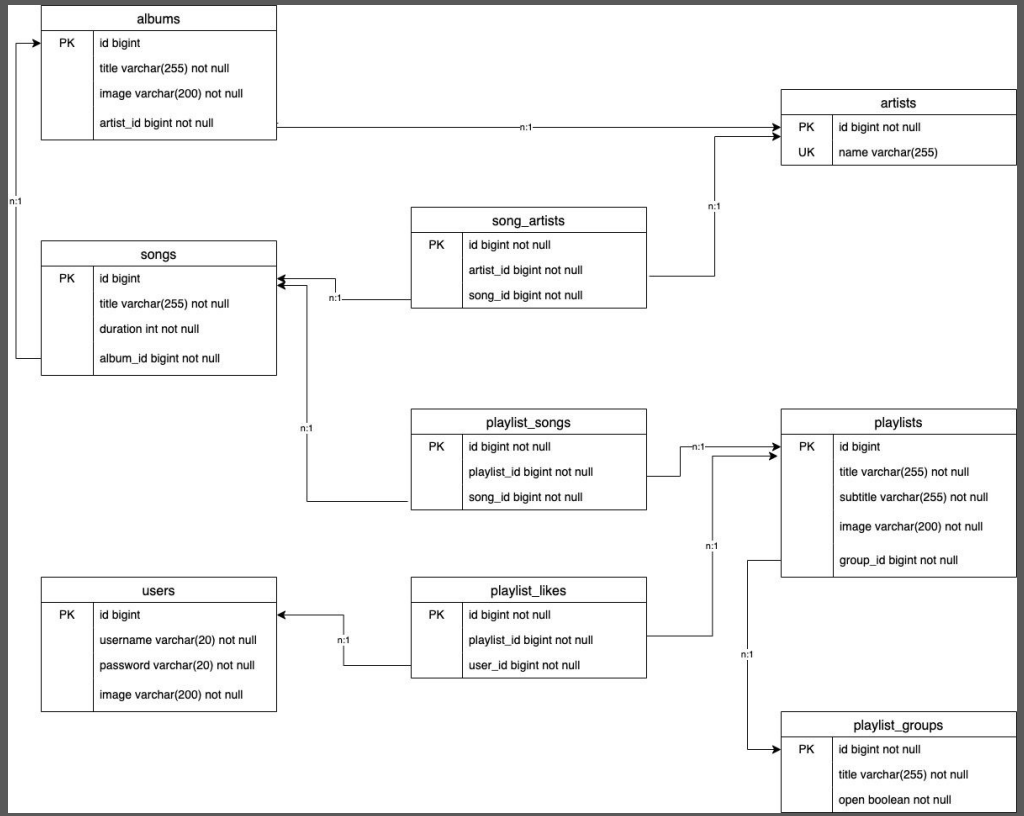
artists

id	name
1	Olivia Rodrigo
2	Paint The Town Red

```
SELECT * FROM artists WHERE id = 1;
```

관계형 데이터베이스 (Relational Database)

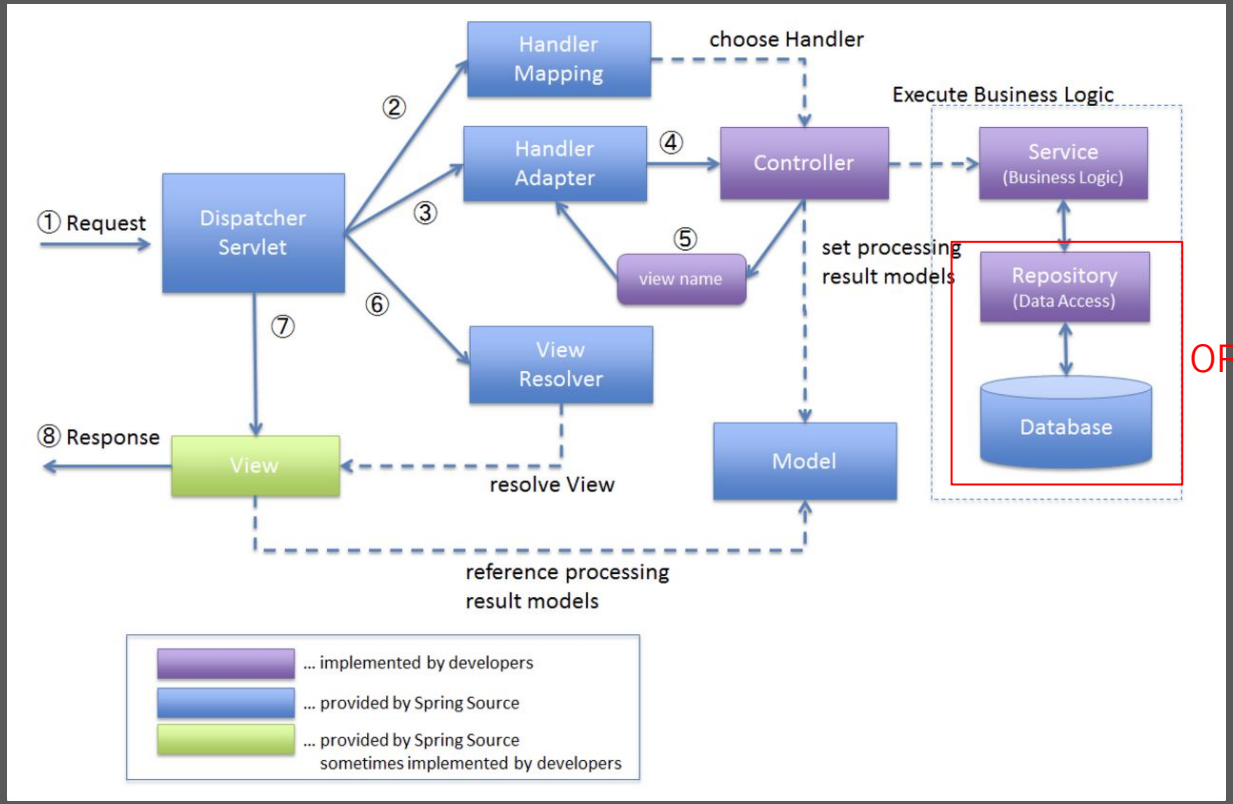
SQL 사전 과제



- 아이디가 1인 아티스트가 발매한 앨범의 제목들
- 제목에 **'Seven'**이 포함된 노래의 제목들
- 제목이 **'GUTS'**인 앨범에 수록된 노래의 제목들
- 제목이 **'Spotify 플레이리스트'**인 플레이리스트 그룹에 속한 플레이리스트의 제목들
- 아이디가 1인 아티스트가 부른 노래의 제목들

ORM (Object Relational Mapping)

객체와 관계형 데이터베이스의 데이터를 자동으로 맵핑



JPA (Java Persistent API)

자바 진영의 ORM 기술 표준

- SQL 중심적 개발으로부터 벗어나, 객체 지향적 개발이 가능
- Hibernate: 대표적인 구현체
- 장점:
  - 생산성 (쿼리보다 비지니스 로직에 집중)
  - 성능 (엔티티 캐시, 쓰기 지연)
  - 벤더 독립성 (MySQL, Oracle, MariaDB..)

JPA

생산성

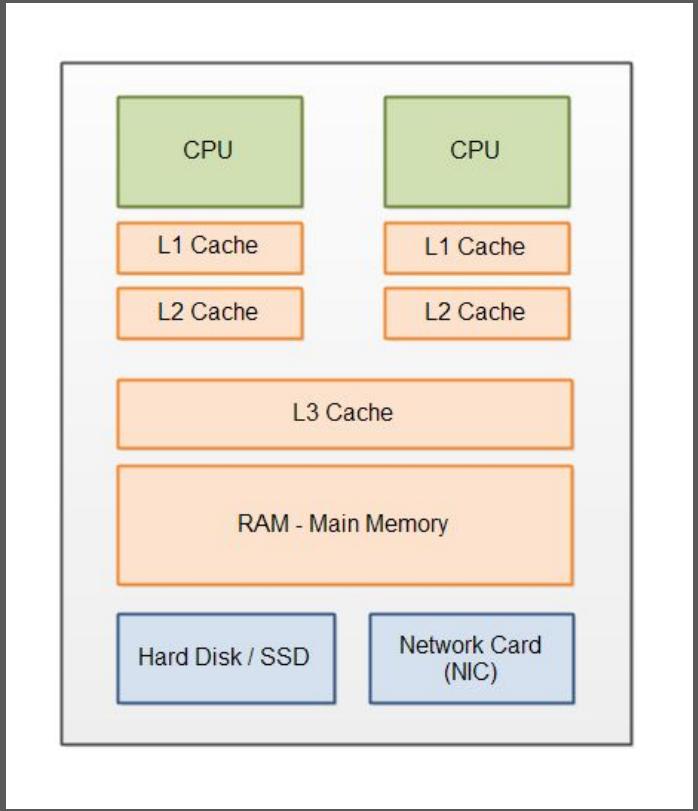
```
interface UserRepository : JpaRepository<UserEntity, Long> {
    fun findByUsername(username: String): UserEntity?
}
```

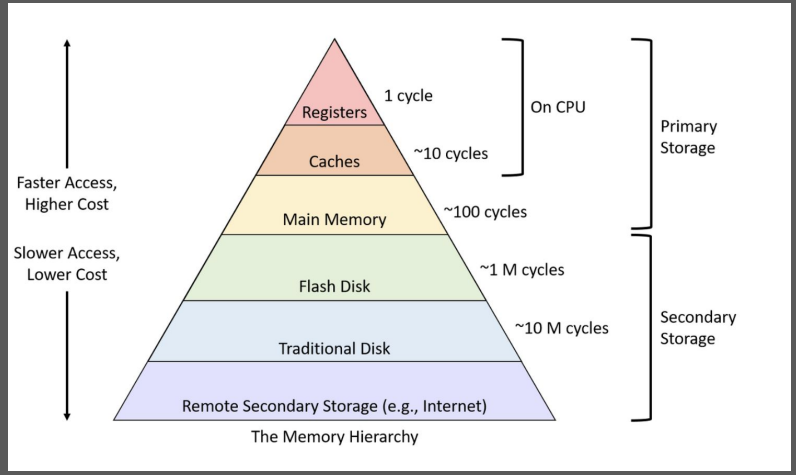
```
Hibernate:
    /* <criteria> */ select
        d1_0.id,
        d1_0.name
    from
        users d1_0
    where
        d1_0.name=?
```

쿼리 자동 생성

JPA

성능 (엔티티 캐시)

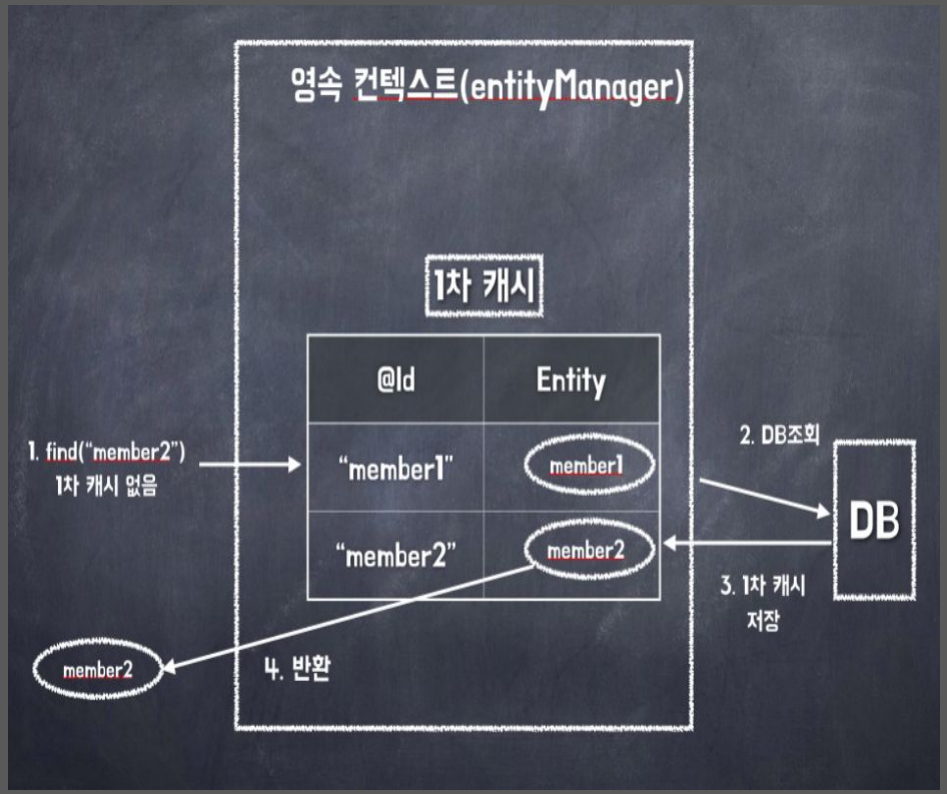




- 온 세상이 캐시이다.
- 캐시 Layer 별 속도 차이는 수십~수백만배

JPA

성능 (엔티티 캐시)



- 네트워크 통신 비용 >>> 메모리 접근 비용
- 엔티티가 1차 캐시에 없으면 엔티티 매니저는 데이터베이스를 조회해서 엔티티를 생성

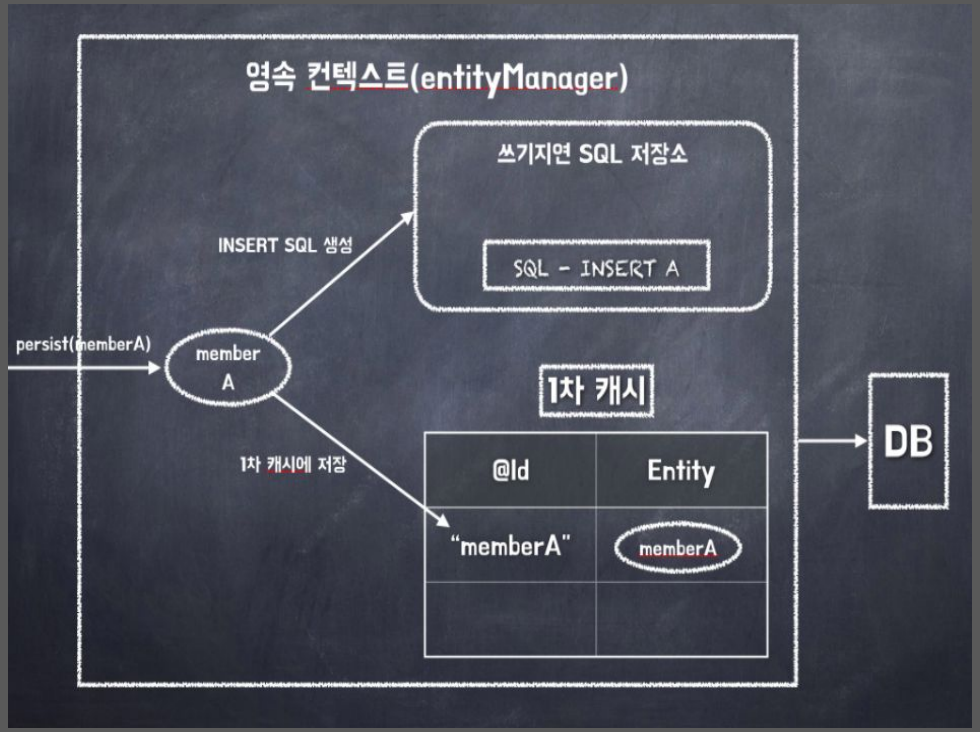
JPA

성능 (엔티티 캐시)

```
@Test
fun `엔티티 캐싱`() {
    val (_ : Unit, queryCount : Int) = queryCounter.count {
        // Hibernate: select a1_0.id,a1_0.name from artists a1_0 where a1_0.id=?
        val artist : ArtistEntity = artistRepository.findById( id: 1L).get()

        println(artist.name)

        // album Id 1이 캐싱되어 있기 때문에 추가 쿼리 X
        val artist2 : ArtistEntity = artistRepository.findById( id: 1L).get()

        println(artist2.name)
    }

    assertThat(queryCount).isEqualTo(1)
}
```

JPA

성능 (쓰기 지연)



- 내부 쿼리 저장소에 INSERT SQL을 차곡차곡 모아둔다.
- 트랜잭션 커밋할 때 모아둔 쿼리를 데이터베이스에 보낸다.

JPA

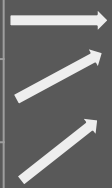
RDB, 테이블 간의 연관 관계

albums	
PK	id bigint
	title varchar(100) not null
	image varchar(200) not null
	artist_id bigint not null

artists	
PK	id bigint not null
UK	name varchar(30)

외래키

id	title	image	artist_id
1	GUTS	http://..	1
2	SOUR	http://..	1
3	Paint The Town Red	http://..	2



id	name
1	Olivia Rodrigo
2	Doja Cat

JPA

테이블과 JPA Entity

albums	
PK	id bigint
	title varchar(100) not null
	image varchar(200) not null
	artist_id bigint not null

```kotlin
@Entity(name = "albums")
class AlbumEntity(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0L,
    val title: String,
    val image: String,
    @ManyToOne 여러 앨범이 한 아티스트에 속할 수 있다 -> ManyToOne
    @JoinColumn(name = "artist_id") 외래 키를 가진 쪽을 연관 관계의 주인이라고 한다
    val artist: ArtistEntity,
)
```

artists	
PK	id bigint not null
UK	name varchar(30)

```kotlin
@Entity(name = "artists")
class ArtistEntity(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long = 0L,
    val name: String,
    @OneToMany(mappedBy = "artist") 한 아티스트가 여러 앨범을 가질 수 있다 -> OneToMany
    val albums: List<AlbumEntity>
)
```

JPA

SQL 쿼리 vs JPA의 객체 탐색 (id가 1인 아티스트와 해당 아티스트의 앨범을 조회)

id	title	image	artist_id
1	GUTS	http://..	1
2	SOUR	http://..	1
3	Paint The Town Red	http://..	2

id	name
1	Olivia Rodrigo
2	Paint The Town Red

```
SELECT albums.*, artists.* FROM artists
    LEFT JOIN albums ON artists.id = albums.artist_id
    WHERE artist_id = 1;
```

```
val artist : ArtistEntity = artistRepository.findById( id: 1L).get()

val albums : List<AlbumEntity> = artist.albums
```

JPA를 통해 SQL문을 신경쓰지 않고,

객체 모델을 이용하여 비즈니스

로직을 구성하는데만 집중할 수 있다

JPA

JPA에서 실제 날리는 쿼리 (의도하지 않은 다량의 쿼리가 발생할 수 있다. 일반적으로 @OneToMany)

```
val artist :ArtistEntity = artistRepository.findById( id: 1L).get() (1)

val albums :List<AlbumEntity> = artist.albums (2)
```

- (1) 시점에 select * from artists where id=1

조인하여 한번에 조회하지 않고 2번의 조회 쿼리가 발생

- (2) 시점에 select * from albums where artist_id=1

JPA

FetchType : EAGER과 LAZY

```
@OneToMany(mappedBy = "artist", fetch = FetchType.EAGER)
val albums: List<AlbumEntity>
```

- EAGER: 연관 객체에 접근하지 않아도 기본적으로 연관 객체를 조회하는 쿼리가 발생
  - @OneToOne, @ManyToOne은 기본적으로 EAGER
  - @OneToOne, @ManyToOne은 EAGER 타입시에 join 쿼리 발생 (쿼리가 한번만 발생한다)
- LAZY: 연관 객체에 접근해야 연관 객체를 조회하는 쿼리가 발생
  - @OneToOne, @ManyToOne은 기본적으로 EAGER

```
val artist :ArtistEntity = artistRepository.findById( id: 1L).get()

val albums :List<AlbumEntity> = artist.albums
```

EAGER 타입일 경우, artist 조회시에 albums 조회 쿼리가 같이 발생한다.

JPA

@OneToOne, @ManyToOne에서의 FetchType.EAGER

```
val album :AlbumEntity = albumRepository.findById( id: 1L).get()
```

- select * from albums left join artists on artists.id=albums.artist_id where al.id=1

- 두 번의 쿼리가 발생하지 않고, 조인으로 한번의 쿼리만 발생

JPA

@OneToMany를 위한 구원, JOIN FETCH (feat. JPQL)

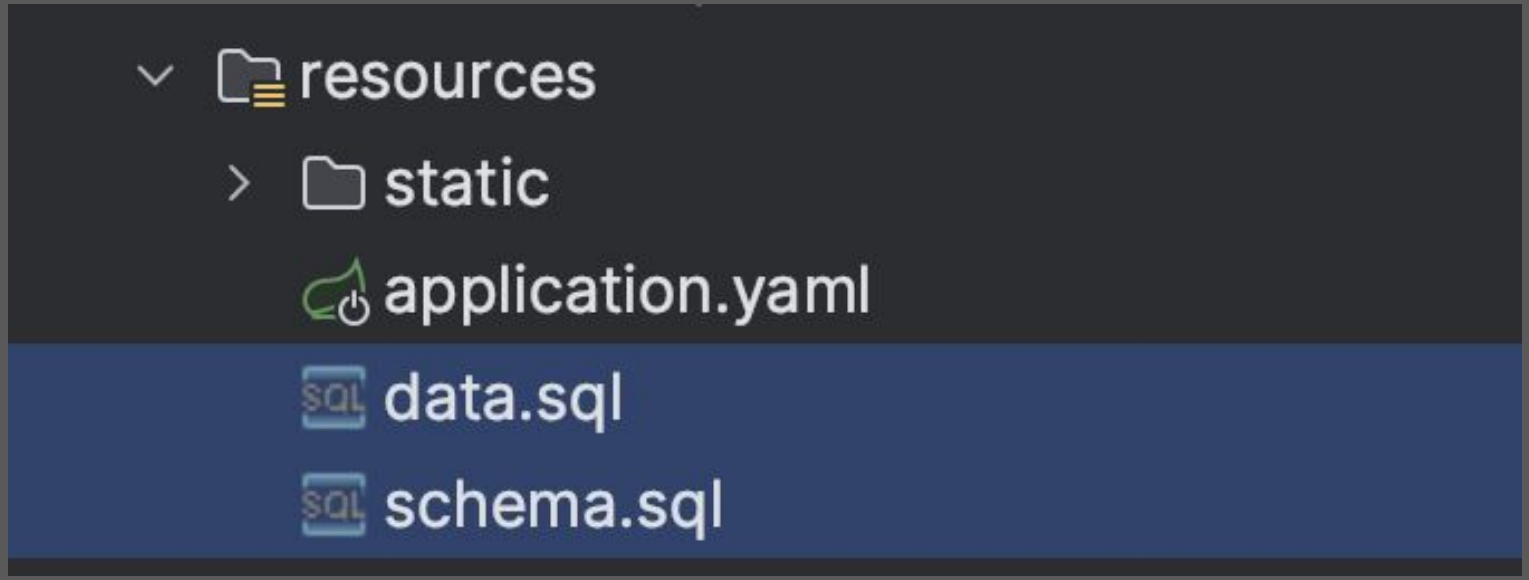
```
interface ArtistRepository : JpaRepository<ArtistEntity, Long> {
    @Query("SELECT a FROM artists a LEFT JOIN FETCH a.albums WHERE a.id = :id")
    fun findByIdWithJoinFetch(id: Long): ArtistEntity?
}
```

```
val artist : ArtistEntity? = artistRepository.findByIdWithJoinFetch( id: 1L)

val albums : List<AlbumEntity> = artist.albums
```

- select * from artists left join albums on artists.id=albums.artist_id where artists.id=1
- 두 번의 쿼리가 발생하지 않고, 조인으로 한번의 쿼리만 발생

Week1 과제 공지

schema.sql, data.sql(H2 임베디드 데이터베이스 데이터 관리)



Week1 과제 공지

application.yaml(서버 설정 관리)

```yaml
spring:
  h2:
    console:
      enabled: true
      path: /h2-console
  datasource:
    url: jdbc:h2:mem:testdb;MODE=MySQL
    driver-class-name: org.h2.Driver
    username: sa
    password: 1234
  jpa:
    defer-datasource-initialization: true
    show-sql: true
```

쿼리 로깅

```
Hibernate:
    /* <criteria> */ select
        d1_0.id,
        d1_0.name
    from
        users d1_0
    where
        d1_0.name=?
```

Week1 과제 공지

@Authenticated

```kotlin
data class User(
    val id: Long,
    val username: String,
    val image: String,
) {
    fun getAccessToken(): String {
        return username.reversed()
    }
}

@Target(AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
annotation class Authenticated
```

```kotlin
override fun resolveArgument(
    parameter: MethodParameter,
    mavContainer: ModelAndViewContainer?,
    webRequest: NativeWebRequest,
    binderFactory: WebDataBinderFactory?,
): User? {
    return runCatching {
        val accessToken : String = requireNotNull(
            webRequest.getHeader( headerName: "Authorization")?.split( ...delimiters: " ")?.get(1)
        )

        userService.authenticate(accessToken) ^runCatching
    }.getOrElse {
        if (parameter.hasParameterAnnotation(Authenticated::class.java)) {
            throw AuthenticateException()
        } else {
            null ^getOrElse
        }
    }
}
```

```kotlin
@GetMapping("/api/v1/playlists/{id}")
fun getPlaylist(
    @PathVariable id: Long,
    user: User?,
): PlaylistResponse {
    val playlist : Playlist = playlistService.get(id)

    val liked : Boolean = if (user == null) {
        false
    } else {
        playlistLikeService.exists(playlistId = id, userId = user.id)
    }

    return PlaylistResponse(playlist, liked)
}

@PostMapping("/api/v1/playlists/{id}/likes")
fun likePlaylist(
    @PathVariable id: Long,
    @Authenticated user: User,
) {
    playlistLikeService.create(playlistId = id, userId = user.id)
}
```

인증이 필수적인 경우
어노테이션으로 명시

Week1 과제 공지

QueryCounter (스레드)

```
@Component
class QueryCounter : StatementInspector {
    data class Result<K>(
        val value: K,
        val queryCount: Int,
    )

    private val isCounting: ThreadLocal<Boolean> = ThreadLocal.withInitial { false }
    private val queryCount: ThreadLocal<Int> = ThreadLocal.withInitial { 0 }

    override fun inspect(sql: String): String {
        if (isCounting.get()) {
            queryCount.set(queryCount.get() + 1)
        }

        return sql
    }

    fun <K> count(block: () -> K): Result<K> {
        isCounting.set(true)
        queryCount.set(0)

        val result : K = block()

        isCounting.set(false)

        return Result(result, queryCount.get())
    }
}
```

스레드 단위로 로컬 변수

Q & A