Spring week 3

유채린
chaerin.du.ub@gmail.com

해당 내용은 인프런 강의를 정리한 내용입니다.
잘못된 내용에 대하여 피드백주시면 감사하겠습니다!

(인프런 아이콘 클릭 시, 참고한 인프런 강의 사이트로 이동합니다.)

Index

스프링 DB 접근

JDBC

- JAVA는 DB랑 붙으려면 JDBC가 무조건 있어야 한다.
  - JDBC를 사용하기 위한 우리의 TO-DO LIST ✨
    - build.gradle 에서 jdbc, DB 관련 라이브러리 추가하기
    - application.properties 에서 스프링 부트 DB 연결 설정 추가하기
    - @Configuration java 파일에서 스프링 설정 변경하기
      - 스프링 부트는 데이터베이스 커넥션 정보를 바탕으로 DataSource(데이터베이스 커넥션을 획득할 때 사용하는 객체)를 생성하고 스프링 빈으로 만들어 둠으로써 DI를 받을 수 있다.

```
@Bean
public MemberRepository memberRepository() {
    return new JdbcMemberRepository(dataSource);
}
```

  - JDBC 기반의 멤버 레퍼지토리를 만들었고 기존 코드는 하나도 손대지 않음
    → OCP 개방 폐쇄 원칙

JDBCTemplate

- **JDBC API 반복 코드를 대부분 제거**해준다. (MyBatis도 동일)
- 순수 JDBC와 동일하게 환경설정 진행하기
- 하지만 **SQL은 직접 작성**해 줘야 한다.

```java
@Override
public Optional<Member> findById(Long id) {
    List<Member> result = jdbcTemplate.query( sql: "select * from member where id = ?", memberRowMapper(), id);
    return result.stream().findAny();
}
```

JPA

- JPA는 기존의 반복 코드는 물론이고, 기본적인 SQL도 JPA가 직접 만들어서 실행해준다.
- SQL과 데이터 중심 설계 → 객체 중심의 설계
- 개발생산성 향상
- JPA를 사용하기 위한 우리의 TO-DO LIST ✨
  - build.gradle 파일에 JPA, DB 관련 라이브러리 추가하기
    (이전에 추가해준 JDBC 라이브러리는 주석 처리하든 제거해준다!! spring-boot-starter-data-jpa 는 내부에 jdbc 관련 라이브러리를 포함)
  - application.properties 파일에 스프링 부트 JPA 설정 추가하기
  - 객체 클래스에 JPA Entity 매핑해주기
  - JPA 객체 리포지토리 추가해주기
  - 서비스 계층에 트랜잭션 추가해주기 (JPA를 통한 모든 데이터 변경은 트랜잭션 안에서 실행해야 함)
  - @Configuration java 파일에서 스프링 설정 변경하기

build.gradle 파일에 JPA, DB 관련 라이브러리 추가하기

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
//  implementation 'org.springframework.boot:spring-boot-starter-jdbc'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    runtimeOnly 'com.h2database:h2'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

application.properties 파일에 스프링 부트 JPA 설정 추가하기

```
spring.datasource.url= jdbc:h2:tcp://localhost/~/test
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none
```

spring.jpa.show-sql=true

➔ JPA가 생성하는 SQL 출력

spring.jpa.hibernate.ddl-auto=none

➔ 예제에서는 테이블 자동생성 기능 끄도록 설정함 (우리는 .. 이미 만들었거든요^^)

➔ create로 변경할 경우, 엔티티 정보를 바탕으로 테이블도 직접 생성해줌

객체 클래스에 JPA Entity 매핑해주기

```
@Entity
public class Member {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name="name")
    private String name;
```

@GeneratedValue(strategy = GenerationType.IDENTITY)

➔ DB에서 PK인 ID 채번 자동으로 해준다.

@Column(name="name")

➔ DB에서도, Entity에서도 이름이 동일(name)하여 굳이 annotation으로 알려줄 필요는 없다!

JPA 객체 리포지토리 추가해주기

```java
public class JpaMemberRepository implements MemberRepository{

    private final EntityManager em;

    public JpaMemberRepository(EntityManager em) {
        this.em = em;
    }
```

JPA는 EntityManager로 모든 게 동작한다.

JPA사용하기 위해 설정했던 build.gradle을 통해 JPA 라이브러리를 받으면 스프링 부트가 자동으로 DB와 연결도 해주고 EntityManager를 만들어준다.
우리는 그럼 injection만 해주면 된다~!

왜 Table이 아니라 Entity라고 하는 걸까?

JPA는 모든 필드를 불러오게 끔 구현되어 있다.
하지만 모든 케이스에서 모든 필드를 다 다루진 않는다.

필드가 ID, NAME, AGE 있다고 할 때,
일부 필드만 사용하는 엔티티 클래스를 하나 더 만들 수 있다.
따라서 테이블은 1개 이지만, 엔티티는 경우에 따라서 여러 개 만들 수 있다.
(JPA를 잘 다룬다면.. 하나의 엔티티로도 충분히 커버할 수 있다고 합니다.)

[Reference]
https://perfectacle.github.io/2018/01/14/jpa-entity-manager-factory/

모든 필드를 사용하는 Entity

```java
@Entity
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false)
    private long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private int age;
```

일부의 필드만 사용하는 Entity

```java
@Entity
@Table(name = "Member")
public class MemberOnlyName {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false)
    private long id;

    @Column(nullable = false)
    private String name;
```

서비스 계층에 트랜잭션 추가해주기

```java
@Transactional
public class MemberService {
```

데이터 저장/변경할 때마다 transactional이 있어야 한다.
JPA는 모든 데이터 변경이 transactional안에서 일어나야 한다!

@Configuration java 파일에서 스프링 설정 변경하기

```java
    @Bean
    public MemberRepository memberRepository() {
        return new JpaMemberRepository(em);
    }
```

스프링 데이터 JPA

- 스프링 부트+JPA ➔ 개발 생산성 향상, 개발 코드 감소

  + 스프링 데이터 JPA ➔ 리포지토리에 구현 클래스 없이 인터페이스만으로도 개발 완료 가능! CRUD 기능도 제공!

- 무조건 JPA 먼저 학습 후 스프링 데이터 JPA 학습하기
- 스프링 데이터 JPA를 사용하기 위한 우리의 TO-DO LIST ✨
  - 앞의 JPA 설정 그대로 사용
  - 스프링 데이터 JPA 회원 리포지토리 추가
  - @Configuration java 파일에서 스프링 설정 변경하기

스프링 데이터 JPA 회원 리포지토리 추가

```java
public interface SpringDataJpaMemberRepository extends JpaRepository<Member, Long>, MemberRepository {

    @Override
    Optional<Member> findByName(String name);
}
```

extends JpaRepository
스프링 데이터 JPA는 JpaRepository를 상속받고 있으면 구현체를 자동으로 만들어줘서 스프링 빈을 자동으로 등록해준다.

스프링 데이터 JPA 제공 기능

1. 인터페이스를 통한 기본적인 CRUD
-> findByName() , findByEmail() 처럼 메서드 이름 만으로 조회 기능 제공

Ex) 스프링 데이터 JPA는 *Optional<Member> findByName(String name);*를 보고,
select m from Member m where m.name = ? 으로 JPQL을 짠다.

2. 페이징 기능 자동 제공

정리하자면

- 순수 JDBC
  어마어마한 쿼리를 개발자가 직접 작성해야 한다.
- 스프링 JdbcTemplate
  반복되는 코드는 줄지만 SQL은 직접 작성해야 한다.
- JPA
  CRUD 작성할 필요는 없지만 SELECT할 때에는 JPQL을 작성해야 한다.
- 스프링 데이터 JPA
  구현 클래스를 작성할 필요없이 인터페이스 구현만으로 끝

실무에서는 JPA와 스프링 데이터 JPA가 기본으로 사용되고 복잡한 동적 쿼리는 Querydsl 라이브러리 사용한다.
이 조합으로 해결 안 되는 쿼리는 JPA가 제공하는 네이티브 쿼리 또는 스프링 JdbcTemplate을 사용한다.

AOP

AOP (Aspect Oriented Programming)

핵심 관심 사항(core concern) vs 공통 관심 사항(cross-cutting concern)

핵심 로직은 건드리지 않고,

AOP 클래스를 따로 선언하여 공통 관심 사항 관리하기!

Ex) 각 메소드 반응 시간 구하기

- AOP를 적용하기 위한 우리의 TO-DO LIST ✨
  - 스프링 빈 등록하기 (AOP class에서 해도 되고, @Configuration java 파일에서 직접 등록해줘도 괜찮다.)
  - AOP 클래스에 공통 코드 작성하기

@Configuration java 파일에 AOP 스프링 빈 등록

```java
@Configuration
public class SpringConfig {

    @Bean
    public TimeTraceApp timeTraceApp() {
        return new TimeTraceApp();
    }
```

빈 순환 참조 에러가 발생한다.
→ 자기 자신을 계속 생성하는 중!!

AOP 클래스에 스프링 빈 등록

```java
@Aspect
@Component
public class TimeTraceApp {

    @Around("execution(* hello.hellospring..*(..))")
    public Object execute(ProceedingJoinPoint joinPoint) throws Throwable {
        ~공통 로직~
        try {
            ~핵심 기능인 메소드들 호출~
        } finally {
            ~공통 로직~
        }
    }
}
```

```java
@Around("execution(* hello.hellospring..*(..)) && !target(hello.hellospring.SpringConfig)")
```

[해결 방법]

1. AOP 클래스에 @Component 어노테이션 추가
   → 코드 자체가 없어 문제 발생하지 않는다.

2. AOP 대상에서 @Configuration java 파일 빼주기
   → Annotation 수정
   @Around("execution(* hello.hellospring..*(..)) && !target(hello.hellospring.SpringConfig)")

AOP 적용하게 되면, 스프링 컨테이너가 스프링 빈 등록할 때, 가짜 스프링 빈(프록시)을 등록한다.
가짜 스프링 빈이 끝나면 진짜 스프링 빈(핵심 기능인 메소드)을 불러준다.

SOLID – 좋은 객체 지향 설계의 5가지 원칙

SOLID

- SRP: 단일 책임 원칙(Single Responsibility Principle)
- OCP: 개방-폐쇄 원칙(Open/Closed Principle)
- LSP: 리스코프 치환 원칙(Liskov Substitution Principle)
- ISP: 인터페이스 분리 원칙(Interface Segregation Principle)
- DIP: 의존관계 역전 원칙(Dependency Inversion Principle)

SRP Single Responsibility Principle
단일 책임 원칙

- 한 클래스는 하나의 책임만
- 기준은 변경
- 수정했는데 파급효과 적으면 단일 책임 원칙 잘 준수함!

OCP Open/Closed Principle

개방-폐쇄 원칙

- 확장에는 열려 있지만 변경에는 닫혀 있어야 한다.

- 다형성 (Polymorphism) 활용해야 한다.

LSP Liskov Substitution Principle
리스코프 치환 원칙

- 프로그램의 정확성을 깨뜨리지 말자.
- 상위 타입을 상속하여 재정의했을 때 프로그램이 깨지면 안된다.
- 재정의를 이상하게 하더라도 컴파일은 잘 된다. 그러니까 컴파일을 넘어서자!!

ISP Interface Segregation Principle
인터페이스 분리 원칙

- 범용 인터페이스 하나 보다는 인터페이스 여러 개를 활용하자
- 비대한 인터페이스보다는 더 작고 구체적인 인터페이스들
- 기능을 적당히 잘 쪼개면 인터페이스도 명확해지고 대체 가능성도 높아진다.

DIP dependency Inversion Principle
의존관계 역전 원칙

- 추상적인 것은 자신보다 구체적인 것에 의존하지 않고, 변화하기 쉬운 것에 의존해서는 안된다.
- 구현 클래스에 의존하지 말고, 인터페이스에 의존하자

감사합니다.