스프링시작하기

스프링 JPA

박명회

1.1 What Is JPA?

Spring Data JPA는 스프링 프레임워크의 일부로서, 데이터 액세스 계층을 구현하는 데 사용되는 도구입니다. JPA(Java Persistence API)는 자바 표준 ORM(Object-Relational Mapping) 기술로, 객체와 관계형 데이터베이스 간의 매핑을 간소화하고 객체 지향 프로그래밍에서 관계형 데이터베이스를 사용하기 위한 표준 방법을 제공합니다.

1. Spring Data JPA는 이러한 JPA 기술을 더 쉽게 사용할 수 있도록 스프링에서 제공하는 프로젝트입니다. Spring Data JPA를 사용하면 개발자는 데이터베이스와의 상호 작용을 단순화할 수 있습니다. 이를 위해 Spring Data JPA는 다음과 같은 주요 기능을 제공합니다:

2. 리포지토리 인터페이스의 자동 구현: Spring Data JPA는 JpaRepository와 같은 리포지토리 인터페이스를 제공하며, 이를 상속받는 인터페이스를 만들면 자동으로 해당 리포지토리의 구현체를 생성합니다. 이를 통해 개발자는 간단한 인터페이스만을 작성하고 실제 구현은 스프링이 자동으로 처리하게 됩니다.

3. 메서드 이름을 통한 쿼리 생성: Spring Data JPA는 메서드 이름을 분석하여 해당 쿼리를 생성합니다. 즉, 개발자는 메서드 이름만으로 데이터베이스 쿼리를 작성할 수 있습니다. 이를 통해 간단한 CRUD(Create, Read, Update, Delete) 기능부터 복잡한 데이터 검색 및 조인 쿼리까지 다양한 쿼리를 작성할 수 있습니다.

4. 쿼리 메서드와 @Query 애너테이션: 메서드 이름으로 작성할 수 없는 복잡한 쿼리는 @Query 애너테이션을 사용하여 직접 쿼리를 정의할 수 있습니다.

5. 페이징 및 정렬: Spring Data JPA는 페이징 및 정렬을 지원하여 데이터베이스에서 데이터를 효율적으로 처리할 수 있습니다.

6. Specification과 QueryDSL: Spring Data JPA는 Specification을 사용하여 동적으로 쿼리를 생성하고, QueryDSL을 사용하여 복잡한 쿼리를 작성할 수 있습니다.

7. Auditing: Spring Data JPA는 엔티티의 생성일, 수정일 등과 같은 감사 정보를 자동으로 관리할 수 있는 기능을 제공합니다.

이러한 기능들을 통해 Spring Data JPA는 개발자가 데이터 액세스 계층을 간소화하고 생산성을 높일 수 있도록 도와줍니다. 데이터베이스와의 상호 작용을 최소화하고 비즈니스 로직에 집중할 수 있도록 해줍니다.

1.2 IOC 컨테이너

JPA와 Oracle 데이터베이스를 사용하는 Spring Boot 애플리케이션을 위한 의존성 및 application.yml 설정

Maven 의존성 (pom.xml)

xml

```xml
<!-- Spring Boot Starter Data JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!-- Oracle JDBC 드라이버 -->
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc8</artifactId>
    <version>19.3.0.0</version>
</dependency>
```

application.yml

yaml Copy code

```yaml
spring:
  datasource:
    url: jdbc:oracle:thin:@<호스트>:<포트>/<SID 또는 서비스 이름>
    username: <사용자명>
    password: <비밀번호>
    driver-class-name: oracle.jdbc.OracleDriver
  jpa:
    database-platform: org.hibernate.dialect.Oracle12cDialect
    hibernate:
      ddl-auto: update
    show-sql: true
```

1.3 프로젝트 생성

User Entity 작성하기

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String email;

    // 생성자, getter, setter 등 필요한 코드 작성
}
```

1.4 초기 데이터 생성

```
spring.jpa.defer-datasource-initialization=true
```

```
spring.sql.init.mode=always
```

```sql
-- data.sql 파일 예제
INSERT INTO user (username, email) VALUES ('user1', 'user1@example.com');
INSERT INTO user (username, email) VALUES ('user2', 'user2@example.com');
```

springjpa 실행하고 나서 data.sql 구문 실행할때 위 구문 application.properties에 넣어야 합니다.

Spring Boot에서는 data.sql 파일을 사용하여 애플리케이션을 시작할 때 자동으로 데이터를 초기화할 수 있습니다. 이 파일은 애플리케이션의 클래스패스 루트에 위치해야 합니다.

아래는 data.sql 파일을 사용하여 초기 데이터를 삽입하는 예제입니다.

그러나 주의할 점은 data.sql 파일은 애플리케이션을 시작할 때마다 실행되기 때문에 이미 데이터가 존재하는 경우 데이터가 중복으로 삽입될 수 있습니다. 따라서 이 방법은 주로 개발 또는 테스트 목적으로 사용되며, 실제 운영 환경에서는 사용되지 않는 것이 좋습니다.

1.4 UserRepository

UserRepository

User 엔티티를 다루기 위한 UserRepository 인터페이스를 작성합니다.

java Copy code

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // 기본적인 CRUD 메서드는 JpaRepository에서 이미 제공됩니다.
}
```

1.4 UserService

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    // 모든 사용자 조회
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    // ID로 사용자 조회
    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }

    // 사용자 추가
    public User addUser(User user) {
        return userRepository.save(user);
    }

    // 사용자 수정
    public User updateUser(Long id, User userDetails) {
        Optional<User> userOptional = userRepository.findById(id);
        if (userOptional.isPresent()) {
            User user = userOptional.get();
            user.setUsername(userDetails.getUsername());
            user.setEmail(userDetails.getEmail());
            return userRepository.save(user);
        } else {
            throw new UserNotFoundException("User not found with id: " + id);
        }
    }

    // 사용자 삭제
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

1.4 UserService

위의 코드에서는 UserService 클래스를 작성하여 UserRepository를 사용하여 데이터베이스에서 User 엔티티를 조작합니다. getAllUsers() 메서드로 모든 사용자를 조회하거나, getUserById() 메서드로 특정 ID의 사용자를 조회하고, addUser() 메서드로 사용자를 추가하며, updateUser() 메서드로 사용자를 수정하고, deleteUser() 메서드로 사용자를 삭제할 수 있습니다.

UserNotFoundException은 사용자를 찾지 못했을 때 발생할 수 있는 예외입니다. 필요에 따라 이 예외를 처리할 수 있습니다.

또한, UserService 클래스에는 @Service 어노테이션을 사용하여 해당 클래스가 비즈니스 로직을 처리하는 서비스 클래스임을 명시했습니다. UserRepository와 마찬가지로 @Repository 어노테이션을 사용하여 UserRepository가 데이터 액세스 관련 작업을 수행하는 레포지토리 인터페이스임을 명시했습니다.

1.4 UserController

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        List<User> users = userService.getAllUsers();
        return new ResponseEntity<>(users, HttpStatus.OK);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        User user = userService.getUserById(id)
                .orElseThrow(() -> new UserNotFoundException("User not found with i
        return new ResponseEntity<>(user, HttpStatus.OK);
    }
```

1.4 UserController

```java
@PostMapping
public ResponseEntity<User> addUser(@RequestBody User user) {
    User newUser = userService.addUser(user);
    return new ResponseEntity<>(newUser, HttpStatus.CREATED);
}

@PutMapping("/{id}")
public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User
    User updatedUser = userService.updateUser(id, userDetails);
    return new ResponseEntity<>(updatedUser, HttpStatus.OK);
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
    userService.deleteUser(id);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

1.4 UserController

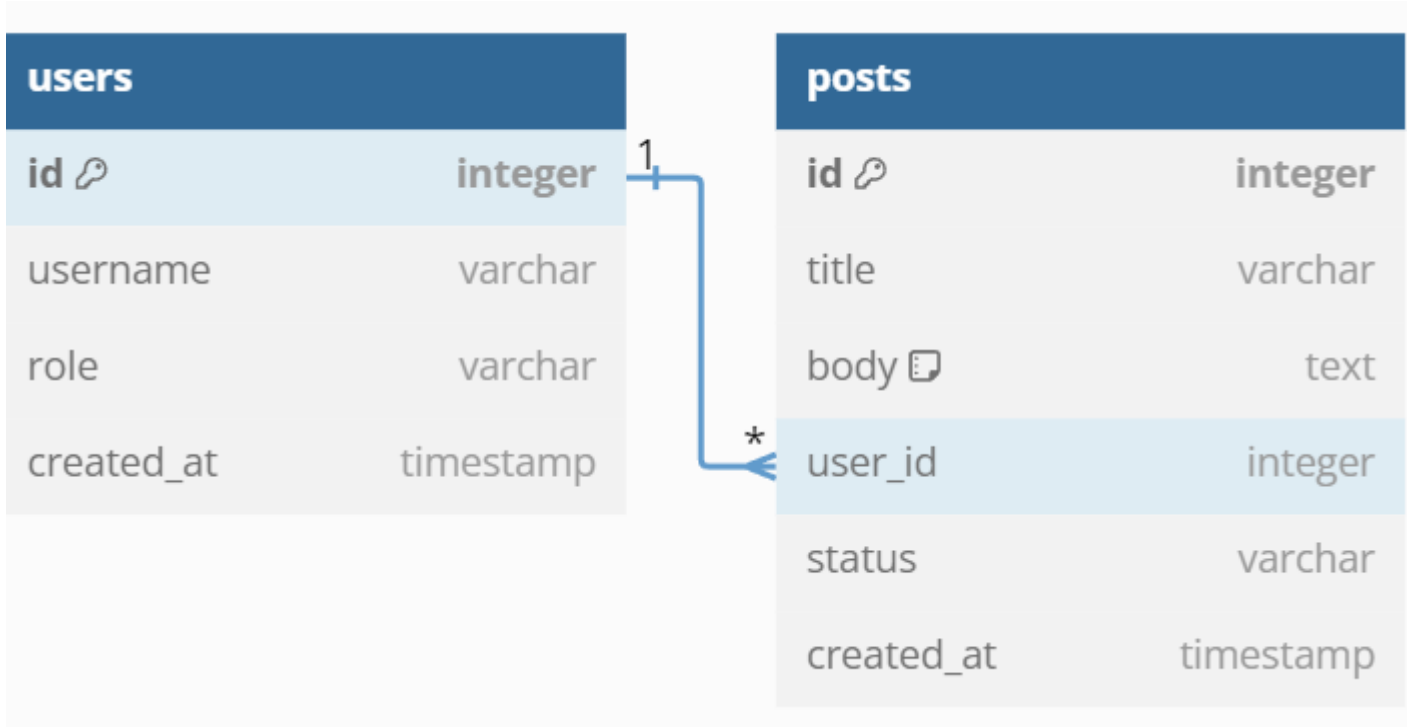
위의 코드에서는 /users 경로에 GET, POST, PUT, DELETE 메서드를 사용하여 사용자 관련 엔드포인트를 정의합니다. 각 메서드는 해당하는 UserService 메서드를 호출하고, 사용자를 조회하거나 추가하거나 수정하거나 삭제합니다.

@RestController 어노테이션은 해당 클래스가 REST 컨트롤러임을 나타내며, @RequestMapping 어노테이션은 컨트롤러에 매핑될 URL을 지정합니다. 그리고 각각의 메서드에는 해당하는 HTTP 메서드와 매핑될 URL을 지정하는 어노테이션을 사용합니다. 예를 들어, @GetMapping("/users")은 GET 메서드를 사용하여 /users 경로에 매핑된 메서드를 나타냅니다.

이렇게 작성된 UserController는 사용자 관련 API를 노출시키고, UserService를 사용하여 해당 API를 처리합니다.

1.4 연관관계

OneToMany(ManyToOne)



1.4 ManyToOne

```java
@Entity
@Getter
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Posts {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // 다대일 관계 설정
    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
}
```

ManyToOne은 JPA(Java Persistence API)에서 사용되는 애노테이션으로, 엔티티 간의 다대일 관계를 매핑하는데 사용됩니다. 다수의 엔티티(Many)가 하나의 엔티티(One)에 속하는 관계를 나타냅니다. 예를 들어, 여러 명의 직원(Employee)이 하나의 부서(Department)에 속할 수 있는 경우가 이에 해당합니다.
ManyToOne 관계를 설정하기 위해서는 다음과 같은 방법을 사용합니다.

@JoinColumn 애노테이션: @JoinColumn 애노테이션은 외래 키(Foreign Key) 매핑을 정의하는데 사용됩니다. name 속성을 사용하여 해당 엔티티 클래스와 매핑할 외래 키의 이름을 지정할 수 있습니다. 이를 통해 다대일 관계를 구현하고 데이터베이스 테이블 간의 관계를 매핑할 수 있습니다.

Fetch 타입: @ManyToOne 애노테이션을 사용할 때 FetchType을 설정하여 엔티티를 로드할 때 Lazy 또는 Eager 방식으로 설정할 수 있습니다. 이는 엔티티를 로드할 때 연관 엔티티를 즉시 가져오는지(Lazy) 또는 즉시 가져오도록 강제하는지(Eager)에 대한 설정입니다.

ManyToOne 관계를 사용하면 다양한 엔티티 간의 관계를 매핑할 수 있으며, 객체 지향적인 방식으로 데이터를 관리할 수 있습니다. 이를 통해 데이터베이스의 관계를 객체 간의 관계로 표현하여 코드의 가독성을 높이고 유지보수를 용이하게 할 수 있습니다.

1.4 OneToMany

```
@OneToMany(mappedBy = "user",cascade = CascadeType.ALL)
private List<Posts> postsList;
```

OneToMany 관계를 설정할 때, @OneToMany 애노테이션을 사용하면서 동시에 JPA에서 OneToMany만 사용할 수 없는 경우는 없습니다. 그러나 몇 가지 주의해야 할 사항이 있습니다.

단방향 관계 vs. 양방향 관계

OneToMany 관계는 단방향 관계로만 설정할 수도 있고, 양방향 관계로 설정할 수도 있습니다.
단방향 관계에서는 @OneToMany 애노테이션만 사용하면 됩니다. 이 경우에는 관계의 주인이 명시되지 않습니다.
양방향 관계에서는 @OneToMany와 함께 mappedBy 속성을 사용하여 관계의 주인을 명시해야 합니다.
Cascade 설정

OneToMany 관계에서는 CascadeType를 적절하게 설정하여 부모 엔티티의 변경이 자식 엔티티에 영향을 주도록 할 수 있습니다. 하지만 Cascade 설정은 선택사항이며, 상황에 따라 필요에 따라 설정합니다.
Fetch 타입

OneToMany 관계에서는 FetchType을 적절히 설정하여 연관된 엔티티를 가져올 때 Eager 또는 Lazy로 설정할 수 있습니다. 이는 관계를 사용하는 시점에 데이터를 어떻게 로드할지 결정하는 옵션입니다.
제약 조건

데이터베이스에서 외래 키(Foreign Key) 제약 조건을 설정할 수 있습니다. 이를 위해선 데이터베이스 스키마에서 제약 조건을 설정해주어야 합니다.
따라서 OneToMany 관계를 설정할 때에도 몇 가지 주의할 사항들이 있지만, OneToMany 관계를 설정하는 것은 가능합니다. 그러나 OneToMany 관계를 설정할 때는 데이터베이스 설계와 객체 간의 관계를 잘 고려하여 설정해야 합니다.

1.4 Querydsl

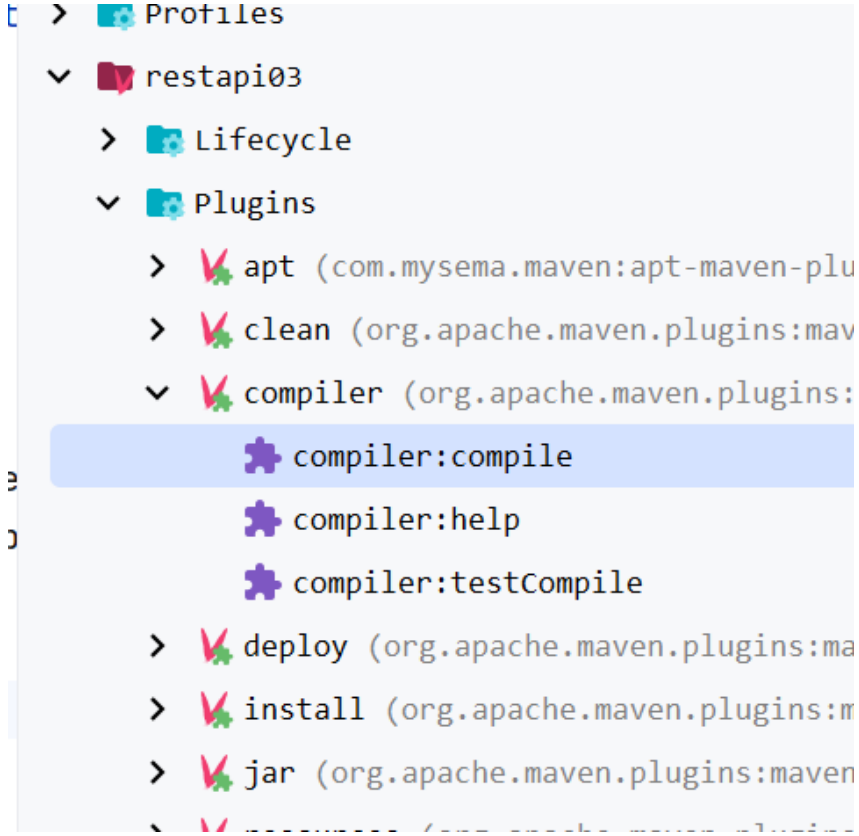
```xml
<properties>
    <java.version>17</java.version>
    <querydsl.version>5.0.0</querydsl.version>
</properties>
```

```xml
<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
    <version>${querydsl.version}</version>
    <classifier>jakarta</classifier>
    <scope>provided</scope>
</dependency>

<dependency>
    <groupId>com.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
    <classifier>jakarta</classifier>
    <version>${querydsl.version}</version>
</dependency>
```

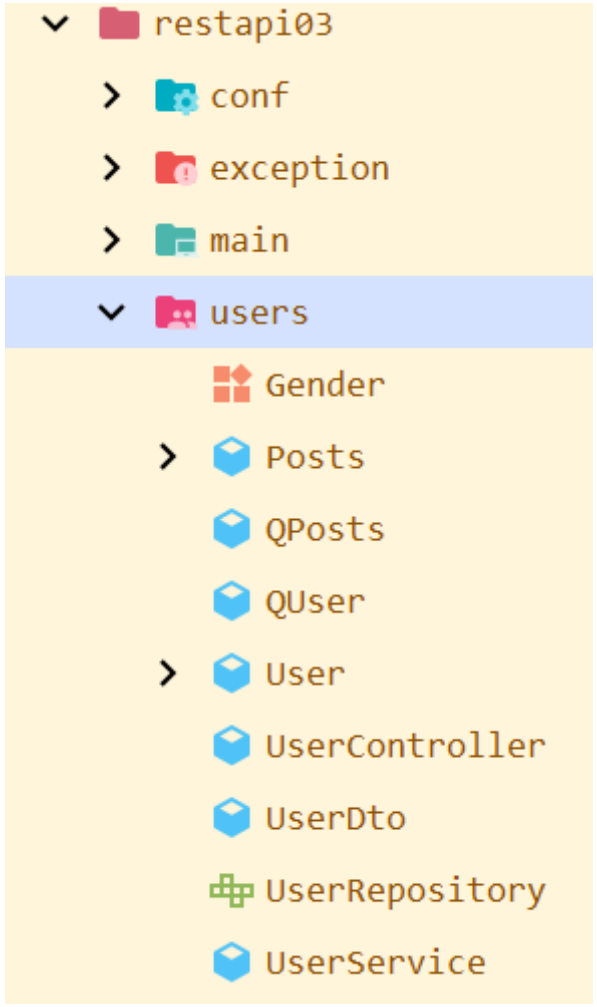
1.4 Querydsl

```
<plugin>
    <groupId>com.mysema.maven</groupId>
    <artifactId>apt-maven-plugin</artifactId>
    <version>1.1.3</version>
    <executions>
        <execution>
            <goals>
                <goal>process</goal>
            </goals>
            <configuration>
                <outputDirectory>target/generated-sources/java</outputDirectory>
                <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
            </configuration>
        </execution>
    </executions>
</plugin>
```



mvn compile 실행

1.4 Querydsl



QClass 생성

1.4 Querydsl

```
@Bean
public JPAQueryFactory jpaQueryFactory(EntityManager em) {
    return new JPAQueryFactory(em);
}
```

```
new *
@GetMapping("qusers")
public ResponseEntity<List<User>> getAllQUsers(){

    QUser user = QUser.user;
    List<User> list = queryFactory.selectFrom(user)
            .fetch();

    if( list.size() ==0 )
        throw new UsersException(ErrorCode.NOTFOUND);
    return ResponseEntity.ok(list);
}
```

1.4 JPQL

JPQL(Java Persistence Query Language)
JPQL은 SQL을 추상화하여 특정 데이터베이스 SQL에 의존적이지 않은 객체지향 쿼리 언어입니다.

테이블을 대상으로 쿼리를 하는것이 아닌 객체(엔티티)를 대상으로 쿼리를 하기에 객체지향 쿼리 언어라고 불립니다.
JPQL은 결국 SQL로 변환되어 데이터베이스에 전달됩니다.

```
1 usage   new *
@Query(value = "select m from User m where m.email = :email")
public User findMyCustom(String email);
```

고맙습니다.