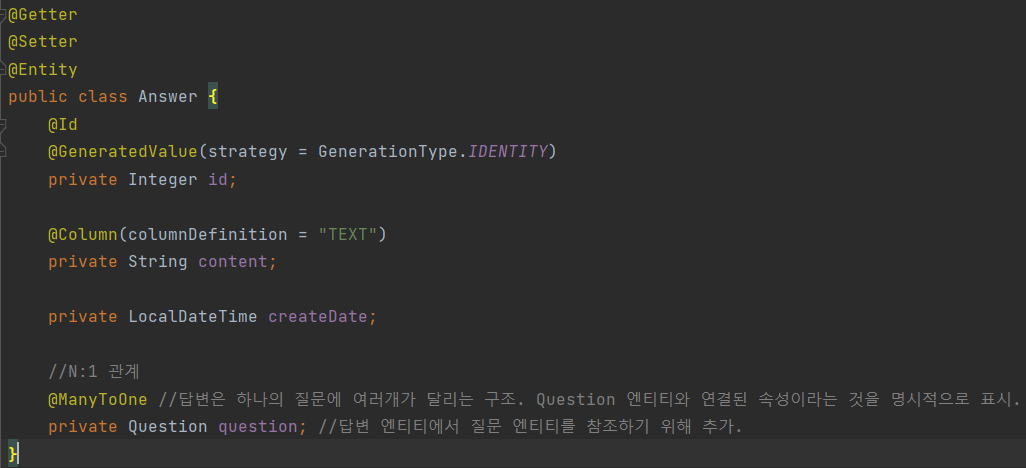
**Entity, 매핑**

데이터베이스와 매핑되는 자바 클래스. DB테이블에 쓰일 속성들과 여러 Entity간의 연관관계를 정의한다. DB테이블 전체가 Entity이고, 각 1개의 행들이 Entity 객체가 되는 것.

Answer.java



**@Entity** 어노테이션으로 이 클래스가 Entity임을 알려준다.

**@Getter** 어노테이션으로 다른 클래스에서 각 필드값을 조회할 수 있는 getter를 해당 클래스에 포함된 모든 멤버 변수에 대해 자동 생성 해준다.

엔티티는 데이터베이스와 바로 연결되어 있으므로 데이터를 자유롭게 변경할 수 있는 **@Setter** 어노테이션은 사용하지 않는 것이 좋다.

**@Id**는 기본 키가 될 값을 지정하는 것.

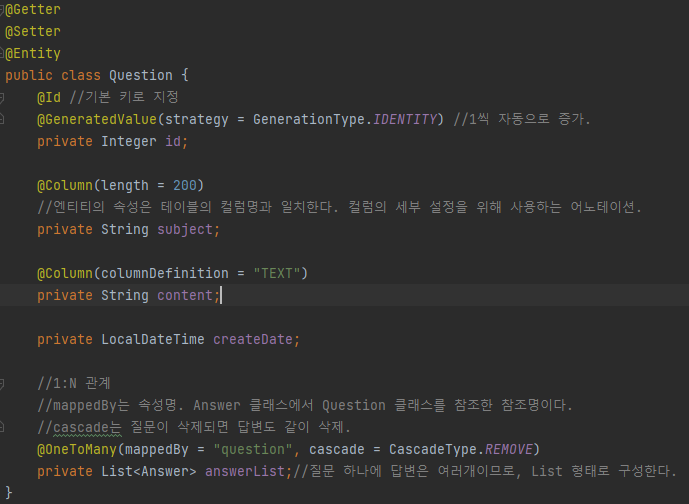
**@GeneratedValue**는 키 값이 자동으로 1씩 증가되는 형태로 설정하는 등이 가능하다.

**@ManyToOne**은 해당 엔티티와 다른 엔티티를 관계 짓고 싶을 때 사용한다. 즉, Answer엔티티의 question 속성과 Question 엔티티가 서로 연결되는 것이다.(실제 DB에선 외래키 관계가 생긴다.)

답변은 하나의 질문에 여러 개가 달릴 수 있는 구조이므로, **답변:질문 = N:1** 이다.

question 속성은 답변 엔티티에서 질문 엔티티를 참조하기 위해 추가했다. 예를 들어 **답변 객체** **(예:answer)를 통해 질문 객체의 제목을 알고 싶다면** answer.getQuestion().getSubject() 처럼 접근할 수 있다.

Question.java



Answer 엔티티에서 Question 엔티티를 참조하는 것은 ManyToOne으로 할 수 있었다. 그렇다면 Question 엔티티에서 Answer 엔티티를 참조하는 방법은 없을까? 가능하다!

답변과 질문의 관계가 N:1이라면, 질문과 답변의 관계는 1:N이라고 할 수 있다. 이런 경우에는 **@OneToMany** 어노테이션을 사용한다. **Question 하나에 Answer는 여러 개이므로 Question 엔티티에 추가할 답변(Answer)의 속성은 List형태로 구성**해야 한다.

이제 질문 객체(예:question)에서 답변을 참조하려면 question.getAnswerList()를 호출하면 된다. @OneToMany 어노테이션에 사용된 **mappedBy**는 참조 엔티티의 속성명을 의미한다. 즉, **Answer 엔티티에서 Question 엔티티를 참조한 속성명 question**을 mappedBy에 전달해야 한다. 이걸 안 해주면 실행 시 중간 테이블이 생성된다

**Repository**

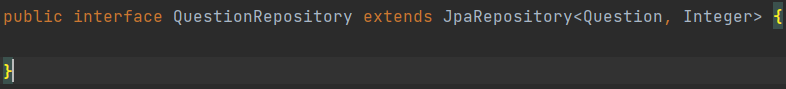
엔티티만으로는 데이터베이스에 데이터를 저장하거나 조회 할 수 없다. **데이터 처리를 위해서는 실제 데이터베이스와 연동하는 JPA 리포지터리가 필요**하다.

리포지터리는 **엔티티에 의해 생성된 데이터베이스 테이블에 접근하는 메서드들(예: findAll, save 등)을 사용하기 위한** **인터페이스**이다. 데이터 처리를 위해서는 테이블에 어떤 값을 넣거나 값을 조회하는 등의 CRUD(Create, Read, Update, Delete)가 필요하다. 이 때 이러한 CRUD를 어떻게 처리할지 정의하는 계층이 바로 리포지터리이다.

AnswerRepository.java 인터페이스

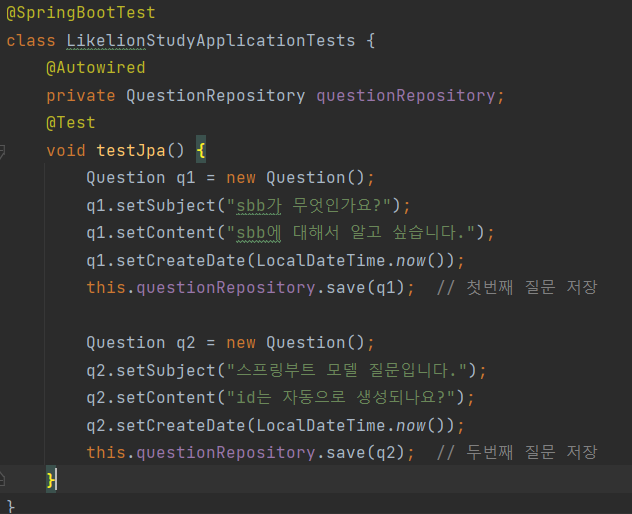


QuestionRepository.java 인터페이스



JpaRepository를 상속할 때는 제네릭스 타입으로 <Question, Integer> 처럼 리포지터리의 대상이 되는 엔티티의 타입(Question)과 해당 엔티티의 기본 키의 속성 타입(Integer)을 지정해야 한다.

**데이터 저장하기 (테스트 파일에서)**

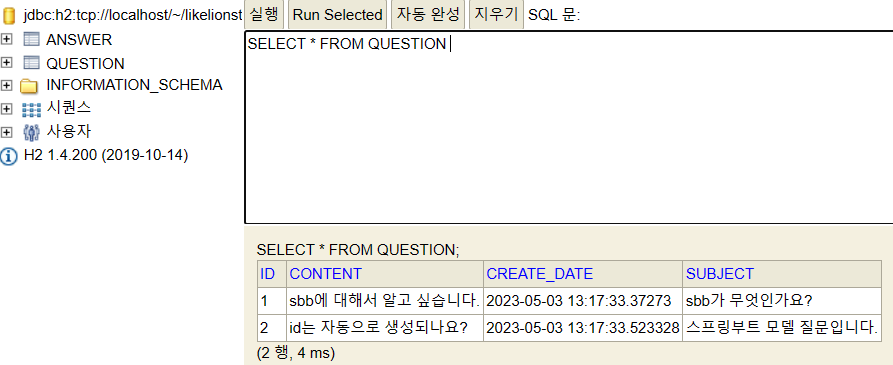


**@SpringBootTest**는 LikelionStudyApplicationTests 클래스가 스프링부트 테스트 클래스임을 의미.

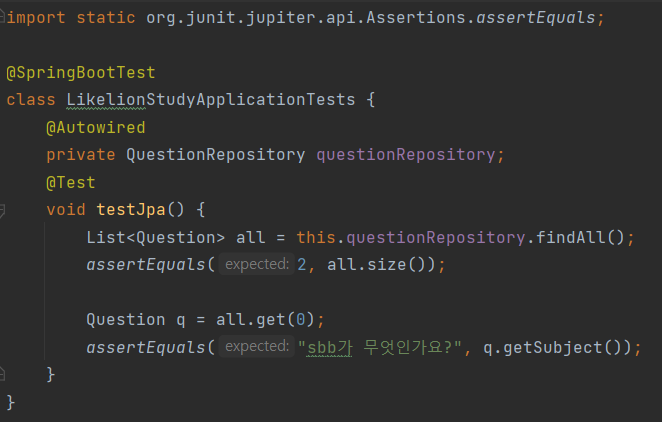
**@Autowired**는 스프링의 DI 기능으로, questionRepository 객체를 스프링이 자동으로 생성해준다.

DI에는 @Autowired 외에도 Setter또는 생성자를 사용하는 방식이 있는데, **생성자를 통한 DI 방식**이 권장된다.

**@Test**는 testJpa() 메서드가 테스트 메서드임을 나타낸다.



**데이터 조회하기**



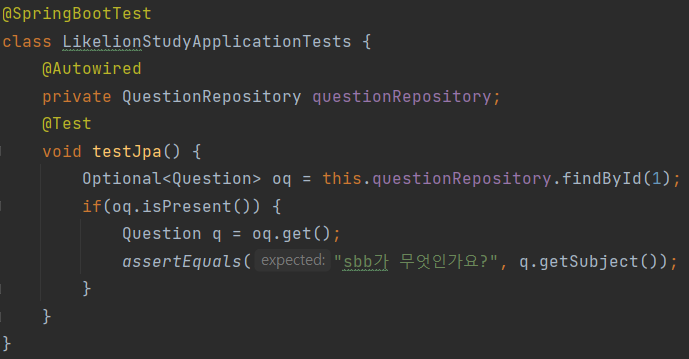
Question 테이블에 저장된 모든 데이터를 조회하기 위해 findAll() 메서드를 사용.

assertEqual은 assertEquals(기대값, 실제값)과 같이 사용하고, 기대값과 실제값이 동일한지를 조사한다. 동일하다면 테스트 성공.



성공!

**ID 값으로 데이터 조회하기.**

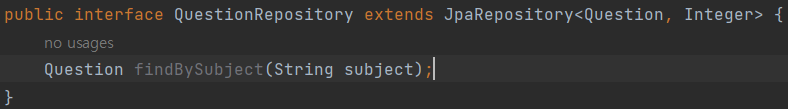


Optional은 null 처리를 유연하게 처리하기 위해 사용하는 클래스로 위와 같이 isPresent로 null이 아닌지를 확인한 후에 get으로 실제 Question 객체 값을 얻어야 한다.

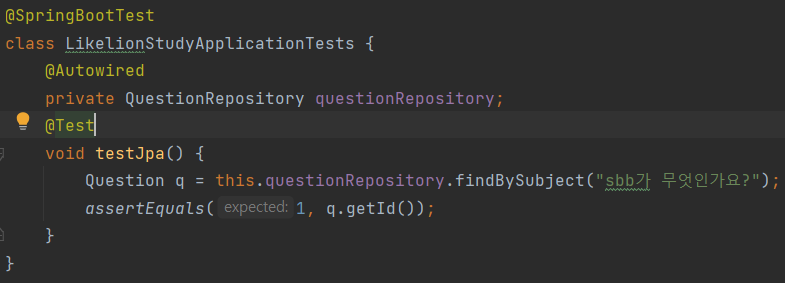
**findBy\*\*\***

이번에는 Question 엔티티의 subject 값으로 데이터를 조회해 보자.

하지만 아쉽게도 Question 리포지터리는 findBySubject와 같은 메서드를 기본적으로 제공하지는 않는다. findBySubject 메서드를 사용하려면 다음처럼 QuestionRepository 인터페이스를 변경해야 한다.



이제는 조회 가능.



인터페이스에 findBySubject 라는 메서드를 선언만 하고 구현은 하지 않았는데 도대체 어떻게 실행이 되는 거지?

이러한 마법은 JpaRepository를 상속한 QuestionRepository 객체가 생성될 때 벌어진다. (DI에 의해 스프링이 자동으로 QuestionRepository 객체를 생성한다. 이 때 프록시 패턴이 사용된다고 한다.) 리포지터리 객체의 메서드가 실행될 때 JPA가 해당 메서드 명을 분석하여 쿼리를 만들고 실행한다.

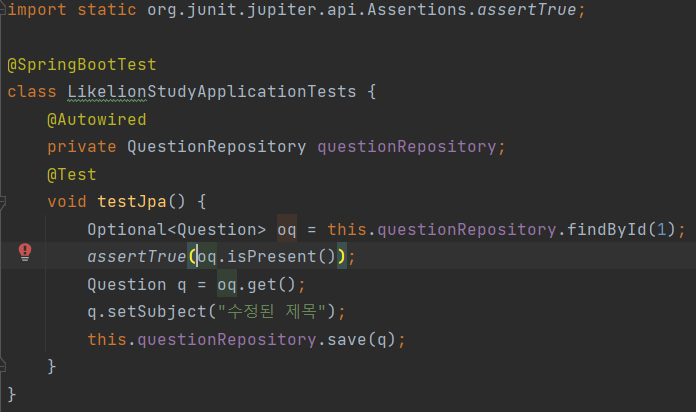
즉, 여러분은findBy + 엔티티의 속성 명(예:findBySubject)과 같은 리포지터리 메서드를 작성하면 해당 속성의 값으로 데이터를 조회할 수 있다.

findBy\*\*\*And\*\*\* 로 두 개의 속성을 And조건으로 조회할 수도 있다.

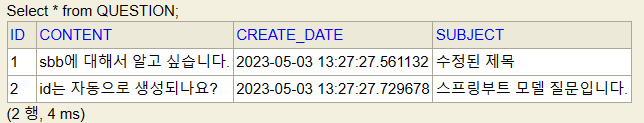
And 말고도, Or, Between, LessThan, Like, In 등등 여러가지로 조회 가능하다.

**데이터 수정하기**

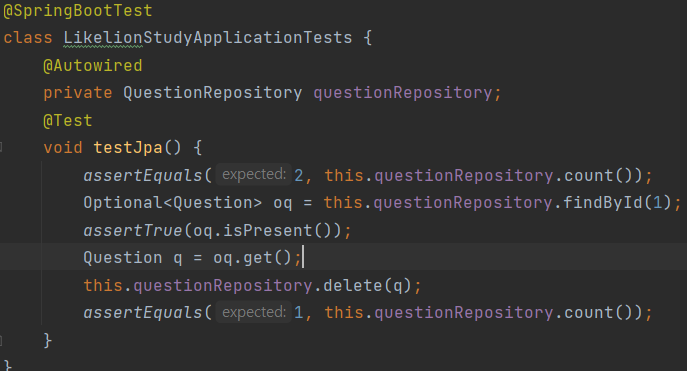
질문 데이터를 수정하는 코드



질문 데이터를 조회한 다음 subject를 “수정된 제목”이라는 값으로 수정했다.

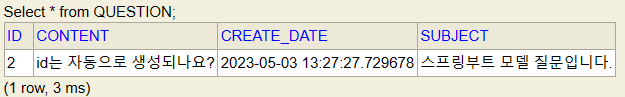


**데이터 삭제하기**

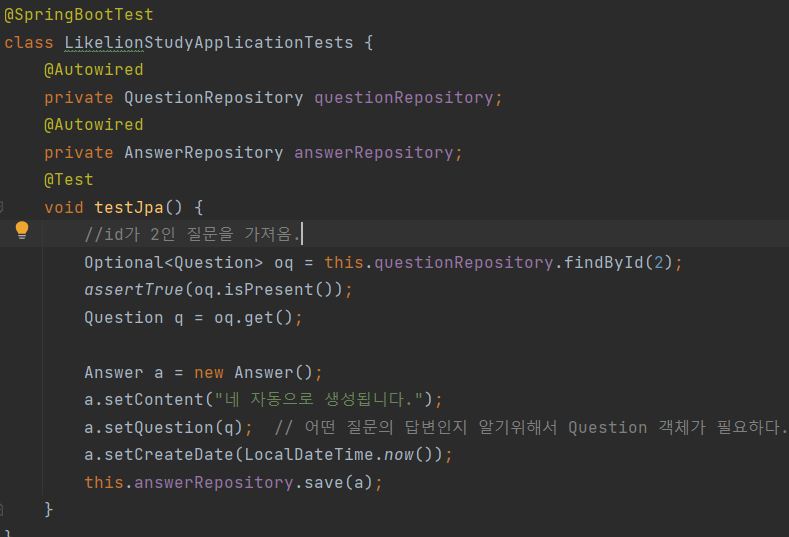


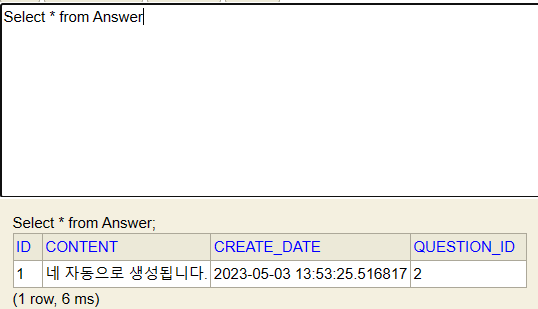
delete()메소드를 이용하여 삭제.

삭제하기 전에는 데이터 수가 2개, 삭제한 후에는 데이터 수가 1개인지 테스트 했다.



**답변 데이터 생성하기**



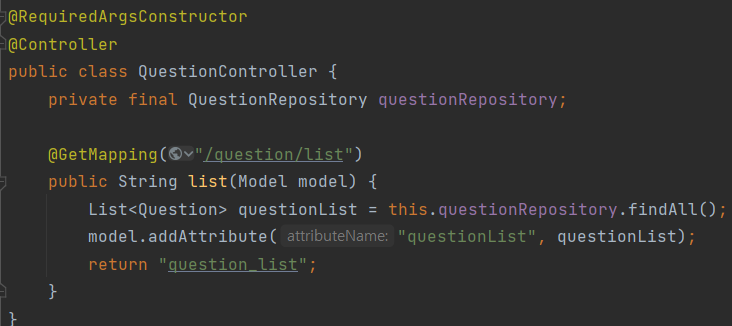


**Controller**

주로 사용자의 요청을 처리하고 난 후, 정해진 뷰에 객체를 넘겨주는 역할을 한다.

**데이터 조회하여 템플릿에 전달하기**

Template에 question\_list.html 파일을 생성해 놓았다.



**@RequiredArgsConstructor** 어노테이션으로 questionRepository 속성을 포함하는 생성자를 생성했다. @RequiredArgsConstrucotr는 롬복이 제공하는 애너테이션으로, final이 붙은 속성을 포함하는 생성자를 자동으로 생성하는 역할을 한다. 따라서 스프링 의존성 주입 규칙에 의해 questionRepository 객체가 자동으로 주입된다.

Question 레포지토리의 **findAll()** 메서드를 이용하여 **questionList를 생성하고 Model 객체에 "questionList" 라는 이름으로 값을 저장**했다. Model 객체는 자바 클래스와 템플릿 간의 연결고리 역할을 한다. **Model 객체에 값을 담아두면 템플릿에서 그 값을 사용할 수 있다.**

**Mapping 요청 방법 (HTTP 요청)**

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @RequestMapping 등등

@GetMapping(“/events/{id}”)  
Public String test(@PathVariable int id){

return 3;

}

처럼 인자로 매핑도 가능.

Spring에서 클래스에 컨트롤러를 지정해주기 위한 어노테이션은 **@Controller**와 **@RestController**가 있다.

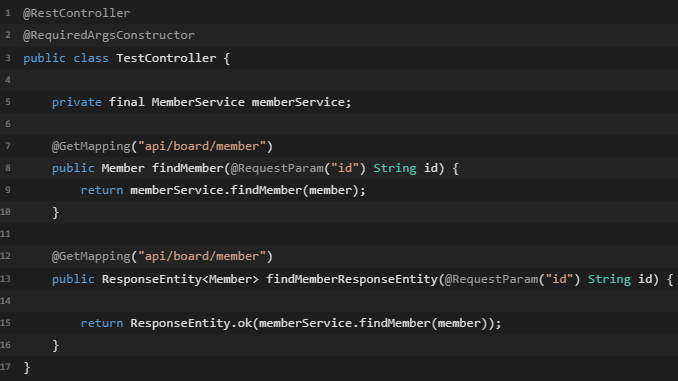
**@Controller**는 주로 View(화면)을 반환하기 위해 사용한다. 객체(Data)를 반환해야 할 경우에는 **@ResponseBody**를 사용하여 JSON 형태의 데이터를 반환할 수도 있다.



상단 코드에서 **findMember()**는 Member이라는 이름을 가진 객체를 ResponseEntity로 감싸서 반환하고, Member를 JSON으로 반환하기 위해 **@ResponseBody** 어노테이션을 붙여주었다.

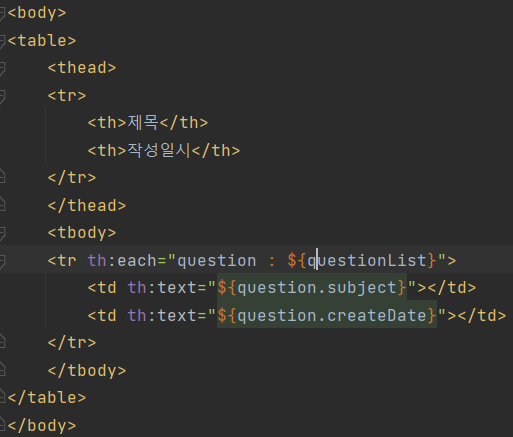
이 2가지를 한 곳에 작성하는 것보다는 데이터를 반환하는 Controller와 View를 반환하는 Controller를 분리해서 작성하는 것이 더욱 좋은 방법이다.

**@RestController**는 @Controller + @ResponseBody가 합쳐진 형태로 JSON 형태의 객체 데이터를 반환한다.



여기서 findMember()는 Member 객체 그대로를 반환한다.

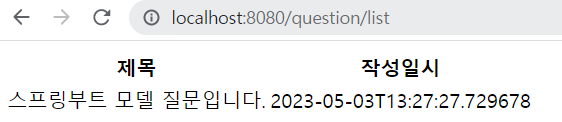
**템플릿에서 전달받은 데이터 사용하기 (Thymeleaf)**



th: 로 시작하는 속성은 타임리프 템플릿 엔진이 사용하는 속성. 이 부분이 자바 코드와 연결된다.

<tr> 엘리먼트를 questionList의 개수만큼 반복해서 question 객체에 대입하여 출력한다. (for-each)

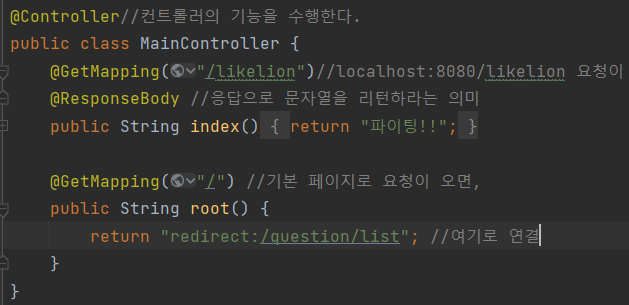
th:text= 에서는 앞의 for문에서 얻은 question 객체를 <td>엘리먼트의 텍스트로 출력한다.



H2 DB에 담긴 question 내용을 가져와 표시했다.

**Root URL**

루트 URL은 <http://localhost:8080> 처럼 도메인명과 포트 뒤에 아무것도 붙이지 않은 URL을 말한다. 우리는 아직 루트 URL에 대한 매핑을 만들지 않았기 때문에 브라우저에서 루트 URL에 접속하면 404 페이지가 나타난다.



root 메서드를 추가하고 / 을 매핑했다. redirect:/question/list는 /question/list URL로 페이지를 리다이렉트 하라는 명령어이다. 이제 <http://localhost:8080> 페이지에 접속하면 root 메서드가 실행되어 질문 목록 페이지가 표시되는 것을 알 수 있다.

**Service**

대부분의 규모 있는 스프링부트 프로젝트는 위에서처럼, **컨트롤러에서 레포지터리를 직접 호출하지 않고 중간에 서비스(Service)를 두어 데이터를 처리**한다. 서비스는 스프링에서 데이터 처리를 위해 작성하는 클래스이다.

**모듈화**

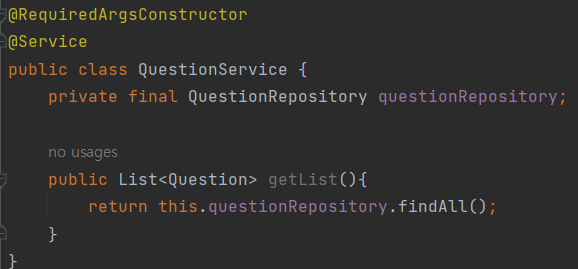
예를 들어 어떤 컨트롤러가 여러 개의 레포지터리를 사용하여 데이터를 조회한 후 가공하여 리턴 한다고 가정해 보자. **이러한 기능을 서비스로 만들어 두면, 컨트롤러에서는 해당 서비스를 호출하여 사용**하면 된다. 하지만 **서비스로 만들지 않고 컨트롤러에서 구현하려 한다면 해당 기능을 필요로 하는 모든 컨트롤러가 동일한 기능을 중복으로 구현해야 한다**. 이러한 이유로 서비스는 모듈화를 위해서 필요하다.

**엔티티 객체와 DTO 객체의 변환**

우리가 작성한 Question, Answer 클래스는 엔티티(Entity) 클래스이다. **엔티티 클래스는 데이터베이스와 직접 맞닿아 있는 클래스**이기 때문에 컨트롤러나 타임리프 같은 템플릿 엔진에 전달하여 사용하는 것은 좋지 않다. 컨트롤러나 타임리프에서 사용하는 데이터 객체는 속성을 변경하여 비즈니스적인 요구를 처리해야 하는 경우가 많은데 엔티티를 직접 사용하여 속성을 변경한다면 테이블 컬럼이 변경되어 엉망이 될 수도 있기 때문이다.

이러한 이유로 Question, Answer 같은 **엔티티 클래스는 컨트롤러에서 사용할 수 없게끔 설계하는 것이 좋다**. 그러기 위해서는 **Question, Answer 대신 사용할 DTO(Data Transfer Object) 클래스가 필요하다**. 그리고 Question, Answer 등의 엔티티 객체를 DTO 객체로 변환하는 작업도 필요하다. 그러면 **엔티티 객체를 DTO 객체로 변환하는 일은 어디서 처리해야 할까? 그렇다. 바로 서비스**이다. 서비스는 컨트롤러와 레포지터리의 중간자적인 입장에서 **엔티티 객체와 DTO 객체를 서로 변환하여 양방향에 전달하는 역할**을 한다.

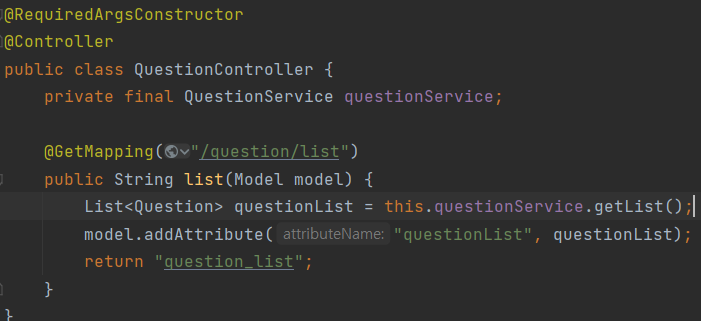
아래 예시에서는 별도의 DTO를 만들지 않고, 엔티티 객체를 컨트롤러와 타임리프에서 그대로 사용할 것이다. 하지만 실무에서는 엔티티 클래스를 대신할 DTO 클래스를 만들어 사용해야 한다.



**@Service** 어노테이션은 서비스로 사용할 클래스라는 의미이다.

질문 목록을 조회하는 getList() 메서드도 추가했다.

이제 컨트롤러에서 레포지토리 대신에 서비스를 사용하도록 수정해야 한다.

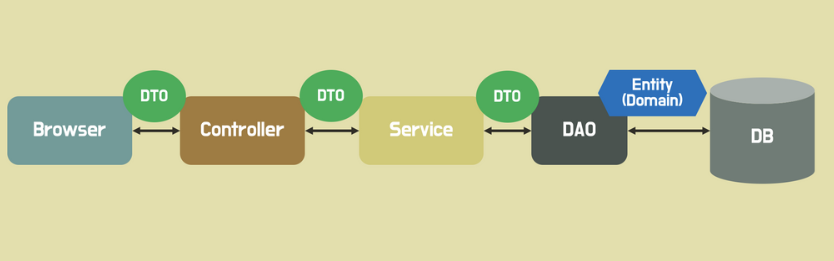


이전에는 컨트롤러에서 QuestionRepository 객체를 직접 사용했었지만, 이제는 QuestionService 객체를 사용하여 레포지토리에 접근한다.

앞으로 작성할 컨트롤러들도 모두 레포지토리를 직접 사용하지 않고, 서비스를 거쳐서 사용할 것이다.

**Controller -> Service -> Repository**

**DTO**



DTO는 **계층 간 데이터 교환을 하기 위해 사용하는 객체**로, DTO는 로직을 가지지 않는 순수한 데이터 객체(Java Beans)이다.

엔티티는 DB와 직접 맞닿아있는 클래스이기 때문에, 컨트롤러나 타임리프 템플릿 엔진에 직접 전달해서 사용하는 것은 보안에 있어서 좋은 방법이 아니다. 따라서 **엔티티 클래스는 컨트롤러에서 사용할 수 없게끔 설계하는 것이 좋다**. 그러기 위해서는 **엔티티 클래스 대신 사용할 DTO(Data Transfer Object) 클래스가 필요하다**.

DTO는 클라이언트 요청에 포함된 데이터를 담아 서버 측에 전달하고, 서버 측의 응답 데이터를 담아 클라이언트에 전달하는 계층간 전달자 역할을 한다.

DTO는 즉, getter/setter 메서드만 가진 클래스를 의미한다.

DB에서 데이터를 얻어서 Service나 Controller 등으로 보낼 때 사용한다.

즉 엔티티를 DTO 형태로 변환한 후 사용한다.