

초급 백엔드 스터디

5주차 - 서비스 계층

지난 주에는...

- JPA 동작 방식
- 레포지토리 계층 작성
- 레포지토리 테스트 작성

이번 주에는...

- 서비스 계층 작성
- 단위 테스트 작성

스프링 Layered Architecture



스프링 Layered Architecture

서비스

- 어플리케이션의 비즈니스 로직이 담기는 계층
- **레포지토리 계층**과 소통하며 **엔티티**, 또는 **DTO**로 소통한다.

서비스 계층

어플리케이션의 **비즈니스 로직**을 수행한다.

<비즈니스 로직 예시>


- 할 일을 생성한다.
- 할 일은 인당 최대 10개까지 생성할 수 있다. (예시, 실제로 구현 x)
- 할 일을 수정한다.
- 할 일은 그 일을 생성한 유저만 삭제할 수 있다.

비즈니스 로직 예시

- 할 일은 인당 최대 10개까지 생성할 수 있다.
1. 할 일을 생성하려는 유저를 조회한다.
 2. 해당 유저가 생성한 모든 할 일을 조회한다.
 3. 모든 할 일의 개수가 10개보다 적은 지 확인한다.
 4. 10개보다 적다면, 할 일을 생성한다.

비즈니스 로직 예시

- 만약 중간에 에러가 발생한다면?

1. 할 일을 생성하려는 유저를 조회한다.  유저가 존재하지 않는다면?
2. 해당 유저가 생성한 모든 할 일을 조회한다.
3. 모든 할 일의 개수가 10개보다 적은 지 확인한다.
4. 10개보다 적다면, 할 일을 생성한다.

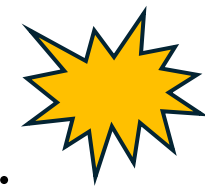
비즈니스 로직 예시

- 서비스 정책에 맞지 않는다면?

1. 할 일을 생성하려는 유저를 조회한다.

2. 해당 유저가 생성한 모든 할 일을 조회한다.

3. 모든 할 일의 개수가 10개보다 적은 지 확인한다.



10개 이상이라면?

4. 10개보다 적다면, 할 일을 생성한다.

비즈니스 로직 예시

- 복잡한 비즈니스 로직

1. A 데이터를 생성한다.
2. B 데이터를 생성한다.  B 데이터를 만들다가 에러 발생
3. A, B 데이터를 이용하여 C 데이터를 생성한다.

- 1번 과정에서 이미 생성된 A 데이터는 어떻게 해야 할까?

비즈니스 로직 예시

1. A 데이터를 생성한다.
 2. B 데이터를 생성한다.
 3. A, B 데이터를 이용하여 C 데이터를 생성한다.
- 세 로직은 셋 다 실행되거나, 셋 다 실행되지 않아야 한다.
 - 이 전체 로직은 더 이상 쪼갤 수 없는 **원자성을 가진 로직**이다.

서비스 계층

- 서비스 계층의 하나의 메서드에는 원자성을 갖는 로직을 기술한다.
- 로직의 원자성을 보장하기 위해서
서비스 계층에 메서드 단위로 트랜잭션을 적용해준다.

JPA 동작 복습

- 트랜잭션 중간에 에러가 발생하면 변경 사항을 롤백한다.



서비스 계층

- todo 패키지에 TodoService 클래스 추가



서비스 계층

- 서비스도 객체를 중복해서 생성할 필요가 없기 때문에 **@Service** 어노테이션을 이용하여 **빈으로 등록**해서 사용한다.

```
no usages
@Service
public class TodoService {

}
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Service {

    | Alias for Component.value.
    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

서비스 계층

- 서비스 계층은 레포지토리 계층에 의존한다.
- **생성자 주입** 방식으로 의존성을 주입받자.

```
@Service
@RequiredArgsConstructor
public class TodoService {

    private final TodoRepository todoRepository;

}
```


할 일 생성

- TodoRepository의 save() 기능을 이용하여 할 일 생성 비즈니스 로직을 작성해보자.

no usages

```
public void createTodo() {  
    Todo todo = new Todo( content: "content", isChecked: false, new User());  
    todoRepository.save(todo);  
}
```

할 일 생성

- 레포지토리 계층을 사용하는 서비스 로직은 반드시 트랜잭션 안에서 동작하도록 작성한다.
- 메서드에 **@Transactional** 을 붙여준다.

```
@Transactional
public void createTodo() {
    Todo todo = new Todo( content: "content", isChecked: false, new User());
    todoRepository.save(todo);
}
```

할 일 생성

- content, user 속성을 하드코딩 하는 것이 맞을까?
- isChecked 필드는 기본값이 false 인데, 매번 명시해야 할까?

```
@Transactional
public void createTodo() {
    Todo todo = new Todo( content "content", isChecked: false, new User());
    todoRepository.save(todo);
}
```

할 일 생성 - 리팩토링

- content, user는 클라이언트가 제공하는 정보이므로 **컨트롤러로부터 받아서 저장하도록** 수정한다.
- 이때 user는 **user_id**만 받은 뒤 DB에서 id를 가지고 **user**를 **조회해서 저장**하자.

```
private final TodoRepository todoRepository;
private final UserRepository userRepository;

no usages
public void createTodo(String content, Long userId) {
    User user = userRepository.findById(userId);
    Todo todo = new Todo(content, isChecked: false, user);
    todoRepository.save(todo);
}
```

할 일 생성 - 리팩토링

- isChecked 속성은 todo 객체를 생성할 때 false가 기본값이다.
- 생성자에서 isChecked 매개변수를 제거하고, 기본값을 지정하자.

```
@Column(name = "todo_is_check", columnDefinition = "tinyint(1)")  
private boolean isChecked = false;
```

```
public Todo(String content, User user) {  
    this.content = content;  
    this.user = user;  
}
```

할 일 생성 - 리팩토링

- isChecked 속성은 todo 객체를 생성할 때 false가 기본값이다.
- 생성자에서 isChecked 매개변수를 제거하고, 기본값을 지정하자.

```
public void createTodo(String content, Long userId) {  
    User user = userRepository.findById(userId);  
    Todo todo = new Todo(content, user);  
    todoRepository.save(todo);  
}
```

할 일 생성 - 예외 처리

- 만약 user_id 값이 존재하지 않으면, findById()의 반환 값은 **null**
- 로그인한 유저만 할 일을 생성할 수 있도록, **user 값이 null 이라면 예외를 발생시킨다.**

```
no usages
public void createTodo(String content, Long userId) throws Exception {
    User user = userRepository.findById(userId);

    if (user == null) {
        throw new Exception("존재하지 않는 유저 ID 입니다.");
    }

    Todo todo = new Todo(content, user);
    todoRepository.save(todo);
}
```

서비스 테스트

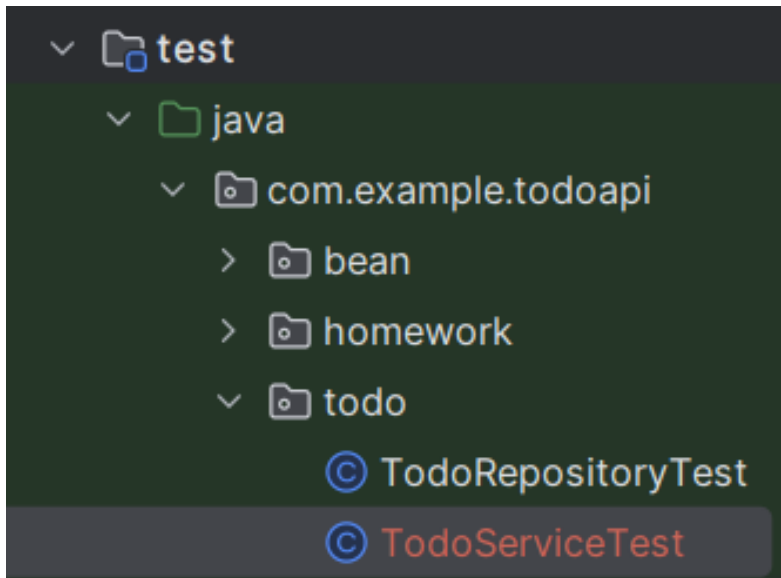
- 할 일 생성 로직이 잘 동작하는지 확인하기 위해 **테스트 코드**를 작성하자.

서비스 테스트

- 서비스 테스트는 **단위 테스트**로 작성해보자.
- 단위 테스트는 스프링 부트와 다른 클래스에 의존하지 않고, 대상 **자바 클래스 하나만 단독으로 테스트하는 것**을 말한다.

서비스 테스트

- test - todo 패키지 아래에 **TodoServiceTest** 클래스를 생성한다.



서비스 테스트

- 서비스 클래스를 단독으로 테스트하기 위해,
서비스가 의존하는 레포지토리는 **가짜 객체**를 사용한다.
- 이 가짜 객체를 가리켜 **mock** 이라고 한다.

서비스 테스트

- 실제 객체를 흉내내는 mock을 만드는 행위를 **mocking** 이라고 한다.
- Mocking을 쉽게 도와주는 Mockito 라이브러리를 사용해 단위 테스트를 작성해보자.

서비스 테스트

- **@ExtendWith** 어노테이션을 사용하여 테스트 클래스에 Mockito를 적용한다.

```
@ExtendWith(MockitoExtension.class)
public class TodoServiceTest {

    |
}
}
```

서비스 테스트

- **@Mock**을 사용하여 주입할 객체를 모킹한다.
- **@InjectMocks**를 사용하여,
서비스 객체를 생성할 때 모킹한 객체를 주입한다.

```
1 usage
@Mock
private TodoRepository todoRepository;

2 usages
@Mock
private UserRepository userRepository;

2 usages
@InjectMocks
private TodoService todoService;
```

서비스 테스트

- **given()** 메서드로 mock 객체의 특정 메서드를 호출했을 때 그 반환값을 임의로 지정할 수 있다.
- **verify()** 메서드로 mock 객체의 특정 메서드를 몇 번 호출했는지 검증할 수 있다.

서비스 테스트

성공 테스트

- 할 일 생성에 성공하면, **todoRepository**의 **save()** 메서드가 1번 호출된다.

```
@Test
void testTodoCreate() throws Exception {
    // given
    given(userRepository.findById(anyLong())).willReturn(new User());

    // when
    todoService.createTodo(content: "content", userId: 1L);

    // then
    verify(todoRepository, times(wantedNumberOfInvocations: 1)).save(any(Todo.class));
}
```


서비스 테스트

예외 테스트

- 존재하지 않는 유저라면 에러가 발생해야 한다.

```
@Test
void testTodoCreate_fail_WhenUserNotExist() throws Exception {
    // given
    given(userRepository.findById(anyLong())).willReturn(value: null);

    // when & then
    Assertions.assertThatThrownBy(() -> todoService.createTodo(content: "content", userId: 1L))
        .isInstanceOf(Exception.class) capture of ?
        .hasMessage("존재하지 않는 유저 ID 입니다.");
}
```

할 일 조회

- 로그인한 유저의 모든 할 일을 조회하는 로직을 작성한다.

```
@Transactional(readonly = true)
public List<Todo> readTodos(Long userId) throws Exception {
    User user = userRepository.findById(userId);

    if (user == null) {
        throw new Exception("존재하지 않는 유저 ID 입니다.");
    }

    return todoRepository.findAllByUser(user);
}
```

조회만 하는 경우,
트랜잭션 내에서
데이터가 변경되지 않도록
readOnly 속성을 활성화한다.

할 일 수정

- 할 일을 수정하는 로직을 작성한다.

```
@Transactional
public void updateContent(Long todoId, String newContent) {
    Todo todo = todoRepository.findById(todoId);
    todo.updateContent(newContent);
}
```

- 엄밀하게 명세를 구현한다면,
조회한 todo가 현재 로그인한 유저의 todo인지 검증해야 한다.

할 일 삭제

- 할 일을 삭제하는 로직을 작성한다.

```
@Transactional
public void deleteContent(Long todoId) {
    todoRepository.deleteById(todoId);
}
```

- 엄밀하게 명세를 구현한다면,
삭제하려는 todo가 현재 로그인한 유저의 todo인지 검증해야 한다.

프로젝트 - 과제

- Todo, User, Friend 에 대한 서비스 계층 코드를 작성하자.
- 서비스 단위 테스트를 작성하자.

프로젝트 명세

Todo mate API 서버 클론 코딩

<주요 기능>

- 유저 회원가입, 로그인
- 로그인한 유저의 할 일 생성 / 조회 / 수정 / 삭제
- 로그인한 유저의 할 일 체크 / 체크 해제
- 다른 유저에게 친구 요청 / 요청 수락 / 친구 조회 / 친구 삭제
- 친구 유저의 할 일 조회