3. 데이터 접근 기술 - 테스트

#1.인강/8.스프링 DB 2/강의#

테스트 - 데이터베이스 연동

데이터 접근 기술에 대해서 더 알아보기 전에 데이터베이스에 연동하는 테스트에 대해서 알아보자. 데이터 접근 기술은 실제 데이터베이스에 접근해서 데이터를 잘 저장하고 조회할 수 있는지 확인하는 것이 필요하다.

지금부터 테스트를 실행할 때 실제 데이터베이스를 연동해서 진행해보자.
앞서 개발한 `ItemRepositoryTest`를 통해서 테스트를 진행할 것이다.

테스트를 실행하기 전에 먼저 지금까지 설정한 `application.properties`를 확인해보자.

main - application.properties

`src/main/resources/application.properties`

```
spring.profiles.active=local
spring.datasource.url=jdbc:h2:tcp://localhost/~/test
spring.datasource.username=sa

logging.level.org.springframework.jdbc=debug
```

test - application.properties

`src/test/resources/application.properties`

```
spring.profiles.active=test
```

테스트 케이스는 `src/test`에 있기 때문에, 실행하면 `src/test`에 있는 `application.properties` 파일이 우선순위를 가지고 실행된다. 그런데 문제는 테스트용 설정에는 `spring.datasource.url`과 같은 데이터베이스

연결 설정이 없다는 점이다.

테스트 케이스에서도 데이터베이스에 접속할 수 있게 test의 `application.properties`를 다음과 같이 수정하자.

test - application.properties 수정

`src/test/resources/application.properties`

```
spring.profiles.active=test
spring.datasource.url=jdbc:h2:tcp://localhost/~/test
spring.datasource.username=sa

logging.level.org.springframework.jdbc=debug
```

테스트 실행 - 로컬DB

`ItemRepositoryTest` 테스트 코드를 확인해보자.
참고로 `src/test` 하위에 있다.

@SpringBootTest

```
@SpringBootTest
class ItemRepositoryTest {}
```

- `ItemRepositoryTest`는 `@SpringBootTest`를 사용한다. `@SpringBootTest`는 `@SpringBootApplication`를 찾아서 설정으로 사용한다.

@SpringBootApplication

```
@Slf4j
//@Import(MemoryConfig.class)
//@Import(JdbcTemplateV1Config.class)
//@Import(JdbcTemplateV2Config.class)
@Import(JdbcTemplateV3Config.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {}
```

- `@SpringBootApplication` 설정이 과거에는 `MemoryConfig.class`를 사용하다가 이제는 `JdbcTemplateV3Config.class`를 사용하도록 변경되었다. 따라서 테스트도 `JdbcTemplate`을 통해 실제 데이터베이스를 호출하게 된다.
- `MemoryItemRepository` ➜ `JdbcTemplateItemRepositoryV3`

테스트 실행

`ItemRepositoryTest` 테스트 전체를 실행하자.

주의! H2 데이터베이스 서버가 실행되어 있어야 한다.

실행 결과

- `updateitem()` : 성공
- `save()` : 성공
- `findItems()` : **실패**

`findItems()` 는 다음과 같은 오류를 내면서 실패했다.

```
java.lang.AssertionError:
Expecting actual:
  [Item(id=7, itemName=ItemTest, price=10000, quantity=10),
    Item(id=8, itemName=itemA, price=10000, quantity=10),
    Item(id=9, itemName=itemB, price=20000, quantity=20),
    Item(id=10, itemName=itemA, price=10000, quantity=10),
...
```

`findItems()` 코드를 확인해보면 상품을 3개 저장하고, 조회한다.

ItemRepositoryTest.findItems()

```
@Test
void findItems() {
    //given
    Item item1 = new Item("itemA-1", 10000, 10);
    Item item2 = new Item("itemA-2", 20000, 20);
    Item item3 = new Item("itemB-1", 30000, 30);

    itemRepository.save(item1);
    itemRepository.save(item2);
    itemRepository.save(item3);

    //여기서 3개 이상이 조회되는 문제가 발생
    test(null, null, item1, item2, item3);
}
```

결과적으로 테스트에서 저정한 3개의 데이터가 조회 되어야 하는데, 기대보다 더 많은 데이터가 조회되었다.

실패 원인

왜 이런 문제가 발생하는 것일까?

혹시 테스트를 실행할 때 `TestDataInit`이 실행되는 것은 아닐까? 이 문제는 아니다. `TestDataInit`은 프로필이 `local`일때만 동작하는데, 테스트 케이스를 실행할 때는 프로필이 `spring.profiles.active=test`이기 때문에 초기화 데이터가 추가되지는 않는다.

문제는 H2 데이터베이스에 이미 과거에 서버를 실행하면서 저장했던 데이터가 보관되어 있기 때문이다. 이 데이터가 현재 테스트에 영향을 준다.

H2 데이터베이스 콘솔을 열어서 데이터를 확인해보자.

H2 데이터베이스 데이터 확인

- http://localhost:8082
- `SELECT * FROM ITEM`을 실행하면 이미 서버를 실행해서 확인 할 때의 데이터가 저장되어 있는 것을 확인할 수 있다.

테스트 - 데이터베이스 분리

로컬에서 사용하는 애플리케이션 서버와 테스트에서 같은 데이터베이스를 사용하고 있으니 테스트에서 문제가 발생한다.

이런 문제를 해결하려면 테스트를 다른 환경과 철저하게 분리해야 한다.

가장 간단한 방법은 테스트 전용 데이터베이스를 별도로 운영하는 것이다.

- H2 데이터베이스를 용도에 따라 2가지로 구분하면 된다.
  - `jdbc:h2:tcp://localhost/~/test` local에서 접근하는 서버 전용 데이터베이스
  - `jdbc:h2:tcp://localhost/~/testcase` test 케이스에서 사용하는 전용 데이터베이스

데이터베이스 파일 생성 방법

- 데이터베이스 서버를 종료하고 다시 실행한다.
  - 사용자명은 `sa` 입력
  - JDBC URL에 다음 입력,
  - `jdbc:h2:~/testcase` (최초 한번)
  - `~/testcase.mv.db` 파일 생성 확인

- 이후부터는 `jdbc:h2:tcp://localhost/~/testcase` 이렇게 접속

테이블 생성하기

`testcase` 데이터베이스에도 `item` 테이블을 생성하자.

`sql/schema.sql` 파일 참고

```
drop table if exists item CASCADE;
create table item
(
    id        bigint generated by default as identity,
    item_name varchar(10),
    price     integer,
    quantity  integer,
    primary key (id)
);
```

접속 정보 변경

이제 접속 정보를 변경하자 참고로 `main`에 있는 `application.properties`는 그대로 유지하고, `test`에 있는 `application.properties`만 변경해야 한다.

main - application.properties

`src/main/resources/application.properties`

```
spring.profiles.active=local
spring.datasource.url=jdbc:h2:tcp://localhost/~/test
spring.datasource.username=sa
```

test - application.properties

`src/test/resources/application.properties`

```
spring.profiles.active=test
spring.datasource.url=jdbc:h2:tcp://localhost/~/testcase
spring.datasource.username=sa
```

접속 정보가 `jdbc:h2:tcp://localhost/~/test` ➜ `jdbc:h2:tcp://localhost/~/testcase`로 변경된 것을 확인할 수 있다.

이제 테스트를 실행해보자!

- `findItems()` 테스트만 단독으로 실행해보자
- 처음에는 실행에 성공한다.
- 그런데 같은 `findItems()` 테스트를 다시 실행하면 테스트에 실패한다.

- 테스트를 2번째 실행할 때 실패하는 이유는 `testcase` 데이터베이스에 접속해서 `item` 테이블의 데이터를 확인하면 알 수 있다.
- 처음 테스트를 실행할 때 저장한 데이터가 계속 남아있기 때문에 두번째 테스트에 영향을 준 것이다.
- 이 문제는 `save()` 같은 다른 테스트가 먼저 실행되고 나서 `findItems()`를 실행할 때도 나타난다. 다른 테스트에서 이미 데이터를 추가했기 때문이다. 결과적으로 테스트 데이터가 오염된 것이다.
- 이 문제를 해결하려면 각각의 테스트가 끝날 때 마다 해당 테스트에서 추가한 데이터를 삭제해야 한다. 그래야 다른 테스트에 영향을 주지 않는다.

테스트에서 매우 중요한 원칙은 다음과 같다.

- **테스트는 다른 테스트와 격리해야 한다.**
- **테스트는 반복해서 실행할 수 있어야 한다.**

물론 테스트가 끝날 때 마다 추가한 데이터에 `DELETE SQL`을 사용해도 되겠지만, 이 방법도 궁극적인 해결책은 아니다. 만약 테스트 과정에서 데이터를 이미 추가했는데, 테스트가 실행되는 도중에 예외가 발생하거나 애플리케이션이 종료되어 버려서 테스트 종료 시점에 `DELETE SQL`을 호출하지 못할 수 도 있다! 그러면 결국 데이터가 남아있게 된다.

이런 문제를 어떻게 해결할 수 있을까?

테스트 - 데이터 롤백

트랜잭션과 롤백 전략

이때 도움이 되는 것이 바로 트랜잭션이다.

테스트가 끝나고 나서 트랜잭션을 강제로 롤백해버리면 데이터가 깔끔하게 제거된다.

테스트를 하면서 데이터를 이미 저장했는데, 중간에 테스트가 실패해서 롤백을 호출하지 못해도 괜찮다. 트랜잭션을 커밋하지 않았기 때문에 데이터베이스에 해당 데이터가 반영되지 않는다.

이렇게 트랜잭션을 활용하면 테스트가 끝나고 나서 데이터를 깔끔하게 원래 상태로 되돌릴 수 있다.

예를 들어서 다음 순서와 같이 각각의 테스트 실행 직전에 트랜잭션을 시작하고, 각각의 테스트 실행 직후에 트랜잭션을 롤백해야 한다. 그래야 다음 테스트에 데이터로 인한 영향을 주지 않는다.

```
1. 트랜잭션 시작
```

```
2. 테스트 A 실행
3. 트랜잭션 롤백

4. 트랜잭션 시작
5. 테스트 B 실행
6. 트랜잭션 롤백
```

테스트는 각각의 테스트 실행 전 후로 동작하는 `@BeforeEach`, `@AfterEach`라는 편리한 기능을 제공한다.
테스트에 트랜잭션과 롤백을 적용하기 위해 다음 코드를 추가하자.

테스트에 직접 트랜잭션 추가

```
@SpringBootTest
class ItemRepositoryTest {

    @Autowired
    ItemRepository itemRepository;

    //트랜잭션 관련 코드
    @Autowired
    PlatformTransactionManager transactionManager;
    TransactionStatus status;

    @BeforeEach
    void beforeEach() {
        //트랜잭션 시작
        status = transactionManager.getTransaction(new
DefaultTransactionDefinition());
    }

    @AfterEach
    void afterEach() {
        //MemoryItemRepository 의 경우 제한적으로 사용
        if (itemRepository instanceof MemoryItemRepository) {
            ((MemoryItemRepository) itemRepository).clearStore();
        }
        //트랜잭션 롤백
        transactionManager.rollback(status);
    }
    //...
}
```

- 트랜잭션 관리자는 `PlatformTransactionManager`를 주입 받아서 사용하면 된다. 참고로 스프링 부트는 자동으로 적절한 트랜잭션 매니저를 스프링 빈으로 등록해준다. (앞서 학습한 스프링 부트의 자동 리소스 등록 장

을 떠올려보자.)

- `@BeforeEach` : 각각의 테스트 케이스를 실행하기 직전에 호출된다. 따라서 여기서 트랜잭션을 시작하면 된다. 그러면 각각의 테스트를 트랜잭션 범위 안에서 실행할 수 있다.
  - `transactionManager.getTransaction(new DefaultTransactionDefinition())` 로 트랜잭션을 시작한다.
- `@AfterEach` : 각각의 테스트 케이스가 완료된 직후에 호출된다. 따라서 여기서 트랜잭션을 롤백하면 된다. 그러면 데이터를 트랜잭션 실행 전 상태로 복구할 수 있다.
  - `transactionManager.rollback(status)` 로 트랜잭션을 롤백한다.

테스트를 실행하기 전에 먼저 테스트에 영향을 주지 않도록 `testcase` 데이터베이스에 접근해서 기존 데이터를 깔끔하게 삭제하자.

모든 ITEM 데이터 삭제

```
delete from item
```

그리고 다음 쿼리를 실행해서 데이터가 모두 삭제되었는지 확인하자.

```
SELECT * FROM ITEM
```

ItemRepositoryTest 실행

이제 `ItemRepositoryTest` 의 테스트를 모두 실행해보자. 여러번 반복해서 실행해도 테스트가 성공하는 것을 확인할 수 있다.

테스트 - @Transactional

스프링은 테스트 데이터 초기화를 위해 트랜잭션을 적용하고 롤백하는 방식을 `@Transactional` 애노테이션 하나로 깔끔하게 해결해준다.
이전에 테스트에 트랜잭션과 롤백을 위해 추가했던 코드들을 주석 처리하자.

```
@SpringBootTest
class ItemRepositoryTest {

    @Autowired
```

```
    ItemRepository itemRepository;

    //트랜잭션 관련 코드
/*
    @Autowired
    PlatformTransactionManager transactionManager;
    TransactionStatus status;

    @BeforeEach
    void beforeEach() {
        //트랜잭션 시작
        status = transactionManager.getTransaction(new
DefaultTransactionDefinition());
    }
*/

    @AfterEach
    void afterEach() {
        //MemoryItemRepository 의 경우 제한적으로 사용
        if (itemRepository instanceof MemoryItemRepository) {
            ((MemoryItemRepository) itemRepository).clearStore();
        }
        //트랜잭션 롤백
        //transactionManager.rollback(status);
    }
    //...
}
```

`ItemRepositoryTest` 테스트 코드에 스프링이 제공하는 `@Transactional`를 추가하자.
`org.springframework.transaction.annotation.Transactional`

```
import org.springframework.transaction.annotation.Transactional;

@Transactional
@SpringBootTest
class ItemRepositoryTest {}
```

테스트를 실행하기 전에 먼저 테스트에 영향을 주지 않도록 `testcase` 데이터베이스에 접근해서 기존 데이터를 깔끔하게 삭제하자.

모든 ITEM 데이터 삭제

```
delete from item
```

그리고 다음 쿼리를 실행해서 데이터가 모두 삭제되었는지 확인하자.

```
SELECT * FROM ITEM
```

ItemRepositoryTest 실행

이제 `ItemRepositoryTest`의 테스트를 모두 실행해보자. 여러번 반복해서 실행해도 테스트가 성공하는 것을 확인할 수 있다.

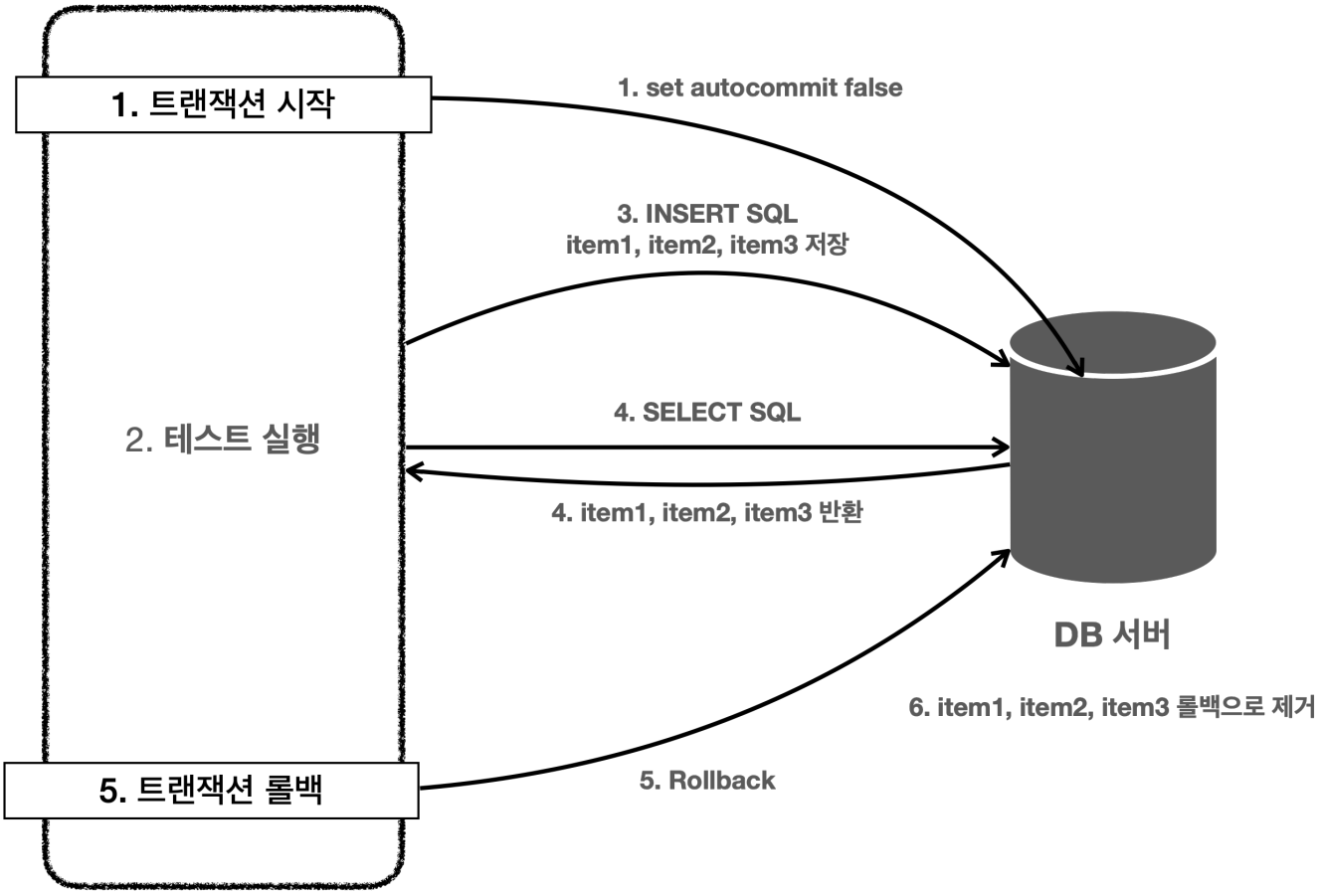
@Transactional 원리

스프링이 제공하는 `@Transactional` 애노테이션은 로직이 성공적으로 수행되면 커밋하도록 동작한다.
그런데 `@Transactional` 애노테이션을 테스트에서 사용하면 아주 특별하게 동작한다.
`@Transactional`이 테스트에 있으면 스프링은 테스트를 트랜잭션 안에서 실행하고, 테스트가 끝나면 트랜잭션을 자동으로 롤백시켜 버린다!

`findItems()`를 예시로 알아보자.

@Transactional이 적용된 테스트 동작 방식

@Transactional이 적용된 테스트



- 1. 테스트에 `@Transactional` 애노테이션이 테스트 메서드나 클래스에 있으면 먼저 트랜잭션을 시작한다.
- 2. 테스트 로직을 실행한다. 테스트가 끝날 때 까지 모든 로직은 트랜잭션 안에서 수행된다.
  - 트랜잭션은 기본적으로 전파되기 때문에, 리포지토리에서 사용하는 JdbcTemplate도 같은 트랜잭션을 사용한다.
- 3. 테스트 실행 중에 INSERT SQL을 사용해서 `item1`, `item2`, `item3`를 데이터베이스에 저장한다.
  - 물론 테스트가 리포지토리를 호출하고, 리포지토리는 JdbcTemplate을 사용해서 데이터를 저장한다.
- 4. 검증을 위해서 SELECT SQL로 데이터를 조회한다. 여기서는 앞서 저장한 `item1`, `item2`, `item3`이 조회되었다.
  - SELECT SQL도 같은 트랜잭션을 사용하기 때문에 저장한 데이터를 조회할 수 있다. 다른 트랜잭션에서는 해당 데이터를 확인할 수 없다.
  - 여기서 `assertThat()`으로 검증이 모두 끝난다.
- 5. `@Transactional`이 테스트에 있으면 테스트가 끝날때 트랜잭션을 강제로 롤백한다.
- 6. 롤백에 의해 앞서 데이터베이스에 저장한 `item1`, `item2`, `item3`의 데이터가 제거된다.

> **참고**
> 테스트 케이스의 메서드나 클래스에 `@Transactional`을 직접 붙여서 사용할 때 만 이렇게 동작한다.
> 그리고 트랜잭션을 테스트에서 시작하기 때문에 서비스, 리포지토리에 있는 `@Transactional`도 테스트에서 시작한 트랜잭션에 참여한다. (이 부분은 뒤에 트랜잭션 전파에서 더 자세히 설명하겠다. 지금은 테스트에서 트랜

> 잭션을 실행하면 테스트 실행이 종료될 때 까지 테스트가 실행하는 모든 코드가 같은 트랜잭션 범위에 들어간다고 이해하면 된다. 같은 범위라는 뜻은 쉽게 이야기해서 같은 트랜잭션을 사용한다는 뜻이다. 그리고 같은 트랜잭션을 사용한다는 것은 같은 커넥션을 사용한다는 뜻이기도 하다.)

정리

- 테스트가 끝난 후 개발자가 직접 데이터를 삭제하지 않아도 되는 편리함을 제공한다.
- 테스트 실행 중에 데이터를 등록하고 중간에 테스트가 강제로 종료되어도 걱정이 없다. 이 경우 트랜잭션을 커밋하지 않기 때문에, 데이터는 자동으로 롤백된다. (보통 데이터베이스 커넥션이 끊어지면 자동으로 롤백되어 버린다.)
- 트랜잭션 범위 안에서 테스트를 진행하기 때문에 동시에 다른 테스트가 진행되어도 서로 영향을 주지 않는 장점이 있다.
- `@Transactional` 덕분에 아주 편리하게 다음 원칙을 지킬수 있게 되었다.
  - 테스트는 다른 테스트와 격리해야 한다.
  - 테스트는 반복해서 실행할 수 있어야 한다.

강제로 커밋하기 - @Commit

`@Transactional`을 테스트에서 사용하면 테스트가 끝나면 바로 롤백되기 때문에 테스트 과정에서 저장한 모든 데이터가 사라진다. 당연히 이렇게 되어야 하지만, 정말 가끔은 데이터베이스에 데이터가 잘 보관되었는지 최종 결과를 눈으로 확인하고 싶을 때도 있다. 이럴 때는 다음과 같이 `@Commit`을 클래스 또는 메서드에 붙이면 테스트 종료후 롤백 대신 커밋이 호출된다. 참고로 `@Rollback(value = false)`를 사용해도 된다.

```
import org.springframework.test.annotation.Commit;

@Commit
@Transactional
@SpringBootTest
class ItemRepositoryTest {}
```

테스트 - 임베디드 모드 DB

테스트 케이스를 실행하기 위해서 별도의 데이터베이스를 설치하고, 운영하는 것은 상당히 번잡한 작업이다. 단순히 테스트를 검증할 용도로만 사용하기 때문에 테스트가 끝나면 데이터베이스의 데이터를 모두 삭제해도 된다. 더 나아가서

테스트가 끝나면 데이터베이스 자체를 제거해도 된다.

임베디드 모드

H2 데이터베이스는 자바로 개발되어 있고, JVM안에서 메모리 모드로 동작하는 특별한 기능을 제공한다. 그래서 애플리케이션을 실행할 때 H2 데이터베이스도 해당 JVM 메모리에 포함해서 함께 실행할 수 있다. DB를 애플리케이션에 내장해서 함께 실행한다고 해서 임베디드 모드(Embedded mode)라 한다. 물론 애플리케이션이 종료되면 임베디드 모드로 동작하는 H2 데이터베이스도 함께 종료되고, 데이터도 모두 사라진다. 쉽게 이야기해서 애플리케이션에서 자바 메모리를 함께 사용하는 라이브러리처럼 동작하는 것이다.

이제 H2 데이터베이스를 임베디드 모드로 사용해보자.

임베디드 모드 직접 사용

임베디드 모드를 직접 사용하는 방법은 다음과 같다.

ItemServiceApplication - 추가

```
package hello.itemservice;

@Slf4j
//@Import(MemoryConfig.class)
//@Import(JdbcTemplateV1Config.class)
//@Import(JdbcTemplateV2Config.class)
@Import(JdbcTemplateV3Config.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ItemServiceApplication.class, args);
    }

    @Bean
    @Profile("local")
    public TestDataInit testDataInit(ItemRepository itemRepository) {
        return new TestDataInit(itemRepository);
    }

    @Bean
    @Profile("test")
    public DataSource dataSource() {
        log.info("메모리 데이터베이스 초기화");
```

```
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:db;DB_CLOSE_DELAY=-1");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }
}
```

- `@Profile("test")`
  - 프로필이 `test`인 경우에만 데이터소스를 스프링 빈으로 등록한다.
  - 테스트 케이스에서만 이 데이터소스를 스프링 빈으로 등록해서 사용하겠다는 뜻이다.
- `dataSource()`
  - `jdbc:h2:mem:db`: 이 부분이 중요하다. 데이터소스를 만들때 이렇게만 적으면 임베디드 모드(메모리 모드)로 동작하는 H2 데이터베이스를 사용할 수 있다.
  - `DB_CLOSE_DELAY=-1`: 임베디드 모드에서는 데이터베이스 커넥션 연결이 모두 끊어지면 데이터베이스도 종료되는데, 그것을 방지하는 설정이다.
  - 이 데이터소스를 사용하면 메모리 DB를 사용할 수 있다.

실행

이제 `ItemRepositoryTest` 테스트를 메모리 DB를 통해 실행해보자.

앞에서 설정은 끝났다. 이제 테스트를 실행만 하면 새로 등록한 메모리 DB에 접근하는 데이터소스를 사용하게 된다.

확실하게 하기 위해서 H2 데이터베이스 서버는 잠시 꺼두자

실행 결과

```
org.springframework.jdbc.BadSqlGrammarException: PreparedStatementCallback; bad
SQL grammar []; nested exception is org.h2.jdbc.JdbcSQLSyntaxErrorException:
Table "ITEM" not found; SQL statement:
insert into item (item_name, price, quantity) values (?, ?, ?) [42102-200]
...
Caused by: org.h2.jdbc.JdbcSQLSyntaxErrorException: Table "ITEM" not found; SQL
statement:
insert into item (item_name, price, quantity) values (?, ?, ?) [42102-200]
```

- 그런데 막상 실행해보면 다음과 같은 오류를 확인 할 수 있다.
- 참고로 오류는 항상 아래에 있는 오류 정보가 더 근본 원인에 가까운 오류 로그이다.
- `Table "ITEM" not found` 이 부분이 핵심이다. 데이터베이스 테이블이 없는 것이다.
- 생각해보면 메모리 DB에는 아직 테이블을 만들지 않았다.

테스트를 실행하기 전에 테이블을 먼저 생성해주어야 한다. 수동으로 할 수도 있지만 스프링 부트는 이 문제를 해결할 아주 편리한 기능을 제공해준다.

스프링 부트 - 기본 SQL 스크립트를 사용해서 데이터베이스를 초기화하는 기능

메모리 DB는 애플리케이션이 종료될 때 함께 사라지기 때문에, 애플리케이션 실행 시점에 데이터베이스 테이블도 새로 만들어주어야 한다.

JDBC나 JdbcTemplate를 직접 사용해서 테이블을 생성하는 DDL을 호출해도 되지만, 너무 불편하다. 스프링 부트는 SQL 스크립트를 실행해서 애플리케이션 로딩 시점에 데이터베이스를 초기화하는 기능을 제공한다.

다음 파일을 생성하자.

위치가 `src/test`이다. 이 부분을 주의하자. 그리고 파일 이름도 맞아야 한다.

`src/test/resources/schema.sql`

```
drop table if exists item CASCADE;
create table item
(
    id        bigint generated by default as identity,
    item_name varchar(10),
    price     integer,
    quantity  integer,
    primary key (id)
);
```

> **참고**
>
> SQL 스크립트를 사용해서 데이터베이스를 초기화하는 자세한 방법은 다음 스프링 부트 공식 메뉴얼을 참고하자.
>
> https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html#howto.data-initialization.using-basic-sql-scripts

실행

`ItemRepositoryTest`를 실행해보면 드디어 테스트가 정상 수행되는 것을 확인할 수 있다.

로그 확인

기본 SQL 스크립트가 잘 실행되는지 로그로 확인하려면 다음이 추가되어 있는지 확인하자.

`src/test/resources/application.properteis`

```
#schema.sql
logging.level.org.springframework.jdbc=debug
```

SQL 스트립트 로그

```
..init.ScriptUtils     : 0 returned as update count for SQL: drop table if
exists item CASCADE
```

```
..init.ScriptUtils      : 0 returned as update count for SQL: create table item
( id bigint generated by default as identity, item_name varchar(10), price
integer, quantity integer, primary key (id) )
```

테스트 - 스프링 부트와 임베디드 모드

스프링 부트는 개발자에게 정말 많은 편리함을 제공하는데, 임베디드 데이터베이스에 대한 설정도 기본으로 제공한다.

스프링 부트는 데이터베이스에 대한 별다른 설정이 없으면 임베디드 데이터베이스를 사용한다.

앞서 직접 설정했던 메모리 DB용 데이터소스를 주석처리하자.

```
package hello.itemservice;

@Slf4j
//@Import(MemoryConfig.class)
//@Import(JdbcTemplateV1Config.class)
//@Import(JdbcTemplateV2Config.class)
@Import(JdbcTemplateV3Config.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ItemServiceApplication.class, args);
    }

    @Bean
    @Profile("local")
    public TestDataInit testDataInit(ItemRepository itemRepository) {
        return new TestDataInit(itemRepository);
    }

/*
    @Bean
    @Profile("test")
    public DataSource dataSource() {
        log.info("메모리 데이터베이스 초기화");
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:db;DB_CLOSE_DELAY=-1");
```

```
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }
*/
}
```

그리고 테스트에서 데이터베이스에 접근하는 설정 정보도 주석처리하자.

test - application.properties

`src/test/resources/application.properties`

```
spring.profiles.active=test
#spring.datasource.url=jdbc:h2:tcp://localhost/~/testcase
#spring.datasource.username=sa

#jdbcTemplate sql log
logging.level.org.springframework.jdbc=debug
```

`spring.datasource.url`, `spring.datasource.username`를 사용하지 않도록 `#`을 사용해서 주석처리 했다.

이렇게 하면 데이터베이스에 접근하는 모든 설정 정보가 사라지게 된다.
이렇게 별다른 정보가 없으면 스프링 부트는 임베디드 모드로 접근하는 데이터소스(`DataSource`)를 만들어서 제공한다. 바로 앞서 우리가 직접 만든 데이터소스와 비슷하다 생각하면 된다.

실행

`ItemRepositoryTest`를 실행해보면 테스트가 정상 수행되는 것을 확인할 수 있다.

참고로 로그를 보면 다음 부분을 확인할 수 있는데 `jdbc:h2:mem` 뒤에 임의의 데이터베이스 이름이 들어가 있다. 이것은 혹시라도 여러 데이터소스가 사용될 때 같은 데이터베이스를 사용하면서 발생하는 충돌을 방지하기 위해 스프링 부트가 임의의 이름을 부여한 것이다.

`conn0: url=jdbc:h2:mem:d8fb3a29-caf7-4b37-9b6c-b0eed9985454`

임베디드 데이터베이스 이름을 스프링 부트가 기본으로 제공하는 `jdbc:h2:mem:testdb`로 고정하고 싶으면 `application.properties`에 다음 설정을 추가하면 된다.

```
spring.datasource.generate-unique-name=false
```

> **참고**
> 임베디드 데이터베이스에 대한 스프링 부트의 더 자세한 설정은 다음 공식 메뉴얼을 참고하자.
> https://docs.spring.io/spring-boot/docs/current/reference/html/

data.html#data.sql.datasource.embedded

정리