1. 데이터 접근 기술 - 시작

#2.인강/8.스프링 DB 2/강의#

- 데이터 접근 기술 진행 방식 소개
- 프로젝트 설정과 메모리 저장소
- 프로젝트 구조 설명1 - 기본
- 프로젝트 구조 설명2 - 설정
- 프로젝트 구조 설명3 - 테스트
- 데이터베이스 테이블 생성
- 정리

데이터 접근 기술 진행 방식 소개

앞으로 실무에서 주로 사용하는 다음과 같은 다양한 데이터 접근 기술들을 학습한다.

적용 데이터 접근 기술

- JdbcTemplate
- MyBatis
- JPA, Hibernate
- 스프링 데이터 JPA
- Querydsl

여기에는 크게 2가지 분류가 있다.

SQLMapper

- JdbcTemplate
- MyBatis

ORM 관련 기술

- JPA, Hibernate
- 스프링 데이터 JPA
- Querydsl

SQL Mapper 주요기능

- 개발자는 SQL만 작성하면 해당 SQL의 결과를 객체로 편리하게 매핑해준다.
- JDBC를 직접 사용할 때 발생하는 여러가지 중복을 제거해주고, 기타 개발자에게 여러가지 편리한 기능을

제공한다.

ORM 주요 기능

- JdbcTemplate이나 MyBatis 같은 SQL 매퍼 기술은 SQL을 개발자가 직접 작성해야 하지만, JPA를 사용하면 기본적인 SQL은 JPA가 대신 작성하고 처리해준다. 개발자는 저장하고 싶은 객체를 마치 자바 컬렉션에 저장하고 조회하듯이 사용하면 ORM 기술이 데이터베이스에 해당 객체를 저장하고 조회해준다.
- JPA는 자바 진영의 ORM 표준이고, Hibernate(하이버네이트)는 JPA에서 가장 많이 사용하는 구현체이다. 자바에서 ORM을 사용할 때는 JPA 인터페이스를 사용하고, 그 구현체로 하이버네이트를 사용한다고 생각하면 된다.
- 스프링 데이터 JPA, Querydsl은 JPA를 더 편리하게 사용할 수 있게 도와주는 프로젝트이다. 실무에서는 JPA를 사용하면 이 프로젝트도 꼭! 함께 사용하는 것이 좋다. 개인적으로는 거의 필수라 생각한다.

궁금한 내용이 많겠지만, 더 자세한 내용은 각각의 기술을 소개하는 장에서 설명하겠다.

데이터 접근 기술 진행 방식

여기에서 설명하는 데이터 저장 기술들은 하나하나 별도의 책이나 강의로 다루어야 할 정도로 내용이 방대하다. 특히 JPA의 경우 스프링과 학습 분량이 비슷할 정도로 공부해야 할 내용이 많다. 그래서 세세한 기능을 설명하기 보다는 주로 해당 기술이 왜 필요한지, 각 기술의 장단점은 무엇인지 설명하는데 초점을 맞추겠다. 그래서 여러분이 필요할 때 해당 데이터 저장 기술을 스스로 학습할 수 있도록 돕는 것이 이번 장의 목표이다.

정리하면 이번 강의의 목표는 다음과 같다.

- 데이터 접근 기술에 대한 기본 이해와 전체 큰 그림을 그린다.
- 각 기술들의 핵심 기능 위주로 학습한다.
- 각 기술들을 점진적으로 도입하는 과정을 통해서 각 기술의 특징과 장단점을 자연스럽게 이해할 수 있다.

먼저 메모리 기반으로 완성되어 있는 프로젝트를 확인하고, 이 프로젝트에 데이터 접근 기술을 하나씩 추가해보자.

프로젝트 설정과 메모리 저장소

스프링 MVC 1편에서 마지막에 완성한 상품 관리 프로젝트를 떠올려보자.
이 프로젝트는 단순히 메모리에 상품 데이터를 저장하도록 되어 있었다.
여기에 메모리가 아닌 실제 데이터 접근 기술들을 하나씩 적용해가면서, 각각의 데이터 접근 기술들을 어떻게 사용하는지, 장단점은 무엇인지 코드로 이해하고 학습해보자.

MVC1 편에서 개발한 상품 관리 프로젝트를 다듬고 일부 기능을 추가해서 `itemservice-db-start` 라는 프로젝트에 넣어두었다.

프로젝트 설정 순서

1. `itemservice-db-start` 의 폴더 이름을 `itemservice-db` 로 변경하자.
2. **프로젝트 임포트**
   File ➡ Open ➡ 해당 프로젝트의 `build.gradle` 을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.
3. `ItemServiceApplication.main()` 을 실행해서 프로젝트가 정상 수행되는지 확인하자.

실행

http://localhost:8080

프로젝트를 실행해서 각각의 기능이 잘 동작하는지 확인해보자.

참고: 해당 프로젝트는 데이터 접근 기술에 초점을 맞추기 위해 검증을 포함한 여러 기능이 빠져있다.

프로젝트 구조 설명1 - 기본

제공되는 프로젝트에 대해서 하나씩 알아보자.

주의! - 스프링 부트 3.0

스프링 부트 3.0을 선택하게 되면 다음 부분을 꼭 확인해주세요.

- **1. Java 17 이상**을 사용해야 합니다.
- **2. javax 패키지 이름을 jakarta로 변경**해야 합니다.
  - 오라클과 자바 라이센스 문제로 모든 javax 패키지를 jakarta로 변경하기로 했습니다.
- **3. H2 데이터베이스를 2.1.214 버전 이상 사용해주세요.**

패키지 이름 변경 예)

- **JPA 애노테이션**
  - javax.persistence.Entity ➡ jakarta.persistence.Entity
- **스프링에서 자주 사용하는 @PostConstruct 애노테이션**
  - javax.annotation.PostConstruct ➡ jakarta.annotation.PostConstruct
- **스프링에서 자주 사용하는 검증 애노테이션**
  - javax.validation ➡ jakarta.validation

스프링 부트 3.0 관련 자세한 내용은 다음 링크를 확인해주세요: https://bit.ly/springboot3

프로젝트 설정

build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.6.5'
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
  id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

ext["hibernate.version"] = "5.6.5.Final"

configurations {
  compileOnly {
      extendsFrom annotationProcessor
  }
}

repositories {
```

```
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- `spring-boot-starter-thymeleaf` : 타임리프 사용
- `spring-boot-starter-web` : 스프링 웹, MVC 기능 사용
- `spring-boot-starter-test` : 스프링이 제공하는 테스트 기능
- `lombok` : lombok을 추가로 테스트에서도 사용하는 설정 주의

도메인 분석

Item

```
package hello.itemservice.domain;

import lombok.Data;

@Data
public class Item {

    private Long id;

    private String itemName;
    private Integer price;
```

```
    private Integer quantity;

    public Item() {
    }

    public Item(String itemName, Integer price, Integer quantity) {
        this.itemName = itemName;
        this.price = price;
        this.quantity = quantity;
    }
}
```

- `Item`은 상품 자체를 나타내는 객체이다. 이름, 가격, 수량을 속성으로 가지고 있다.

리포지토리 분석

ItemRepository 인터페이스

```
package hello.itemservice.repository;

import hello.itemservice.domain.Item;

import java.util.List;
import java.util.Optional;

public interface ItemRepository {

    Item save(Item item);

    void update(Long itemId, ItemUpdateDto updateParam);

    Optional<Item> findById(Long id);

    List<Item> findAll(ItemSearchCond cond);

}
```

- 메모리 구현체에서 향후 다양한 데이터 접근 기술 구현체로 손쉽게 변경하기 위해 리포지토리에

인터페이스를 도입했다.

- 각각의 기능은 메서드 이름으로 충분히 이해가 될 것이다.

ItemSearchCond

```
package hello.itemservice.repository;

import lombok.Data;

@Data
public class ItemSearchCond {

    private String itemName;
    private Integer maxPrice;

    public ItemSearchCond() {
    }

    public ItemSearchCond(String itemName, Integer maxPrice) {
        this.itemName = itemName;
        this.maxPrice = maxPrice;
    }
}
```

- 검색 조건으로 사용된다. 상품명, 최대 가격이 있다. 참고로 상품명의 일부만 포함되어도 검색이 가능해야 한다. (`like` 검색)
- `cond` ➜ `condition` 을 줄여서 사용했다.
  - 이 프로젝트에서 검색 조건은 뒤에 `Cond` 를 붙이도록 규칙을 정했다.

ItemUpdateDto

```
package hello.itemservice.repository;

import lombok.Data;

@Data
public class ItemUpdateDto {
    private String itemName;
    private Integer price;
```

```
    private Integer quantity;

    public ItemUpdateDto() {
    }

    public ItemUpdateDto(String itemName, Integer price, Integer quantity) {
        this.itemName = itemName;
        this.price = price;
        this.quantity = quantity;
    }
}
```

- 상품을 수정할 때 사용하는 객체이다.
- 단순히 데이터를 전달하는 용도로 사용되므로 DTO를 뒤에 붙였다.

DTO(data transfer object)

- 데이터 전송 객체
- DTO는 기능은 없고 데이터를 전달만 하는 용도로 사용되는 객체를 뜻한다.
  - 참고로 DTO에 기능이 있으면 안되는가? 그것은 아니다. 객체의 주 목적이 데이터를 전송하는 것이라면 DTO라 할 수 있다.
- 객체 이름에 DTO를 꼭 붙여야 하는 것은 아니다. 대신 붙여두면 용도를 알 수 있다는 장점은 있다.
- 이전에 설명한 `ItemSearchCond`도 DTO 역할을 하지만, 이 프로젝트에서 `Cond`는 검색 조건으로 사용한다는 규칙을 정했다. 따라서 DTO를 붙이지 않아도 된다. `ItemSearchCondDto` 이렇게 하면 너무 복잡해진다. 그리고 `Cond`라는 것만 봐도 용도를 알 수 있다.
- 참고로 이런 규칙은 정해진 것이 없기 때문에 해당 프로젝트 안에서 일관성 있게 규칙을 정하면 된다.

MemoryItemRepository

```
package hello.itemservice.repository.memory;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import org.springframework.stereotype.Repository;
import org.springframework.util.ObjectUtils;

import java.util.*;
import java.util.stream.Collectors;
```

```java
@Repository
public class MemoryItemRepository implements ItemRepository {

    private static final Map<Long, Item> store = new HashMap<>(); //static
    private static long sequence = 0L; //static

    @Override
    public Item save(Item item) {
        item.setId(++sequence);
        store.put(item.getId(), item);
        return item;
    }

    @Override
    public void update(Long itemId, ItemUpdateDto updateParam) {
        Item findItem = findById(itemId).orElseThrow();
        findItem.setItemName(updateParam.getItemName());
        findItem.setPrice(updateParam.getPrice());
        findItem.setQuantity(updateParam.getQuantity());
    }

    @Override
    public Optional<Item> findById(Long id) {
        return Optional.ofNullable(store.get(id));
    }

    @Override
    public List<Item> findAll(ItemSearchCond cond) {
        String itemName = cond.getItemName();
        Integer maxPrice = cond.getMaxPrice();
        return store.values().stream()
                .filter(item -> {
                    if (ObjectUtils.isEmpty(itemName)) {
                        return true;
                    }
                    return item.getItemName().contains(itemName);
                }).filter(item -> {
                    if (maxPrice == null) {
```

```
                        return true;
                    }
                    return item.getPrice() <= maxPrice;
                })
                .collect(Collectors.toList());
    }

    public void clearStore() {
        store.clear();
    }

}
```

- `ItemRepository` 인터페이스를 구현한 메모리 저장소이다.
- 메모리이기 때문에 자바를 다시 실행하면 기존에 저장된 데이터가 모두 사라진다.
- `save`, `update`, `findById` 는 쉽게 이해할 수 있을 것이다. 참고로 `findById` 는 `Optional` 을 반환해야 하기 때문에 `Optional.ofNullable` 을 사용했다.
- `findAll` 은 `ItemSearchCond` 이라는 검색 조건을 받아서 내부에서 데이터를 검색하는 기능을 한다. 데이터베이스로 보면 `where` 구문을 사용해서 필요한 데이터를 필터링 하는 과정을 거치는 것이다.
  - 여기서 자바 스트림을 사용한다.
  - `itemName` 이나, `maxPrice` 가 `null` 이거나 비었으면 해당 조건을 무시한다.
  - `itemName` 이나, `maxPrice` 에 값이 있을 때만 해당 조건으로 필터링 기능을 수행한다.
- `clearStore()` 메모리에 저장된 `Item` 을 모두 삭제해서 초기화한다. 테스트 용도로만 사용한다.

서비스 분석

ItemService 인터페이스

```
package hello.itemservice.service;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;

import java.util.List;
import java.util.Optional;
```

```
public interface ItemService {

    Item save(Item item);

    void update(Long itemId, ItemUpdateDto updateParam);

    Optional<Item> findById(Long id);

    List<Item> findItems(ItemSearchCond itemSearch);
}
```

- 서비스의 구현체를 쉽게 변경하기 위해 인터페이스를 사용했다.
- 참고로 서비스는 구현체를 변경할 일이 많지는 않기 때문에 사실 서비스에 인터페이스를 잘 도입하지는 않는다.
  - 여기서는 예제 설명 과정에서 구현체를 변경할 예정이어서 인터페이스를 도입했다.

ItemServiceV1

```
package hello.itemservice.service;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
@RequiredArgsConstructor
public class ItemServiceV1 implements ItemService {

    private final ItemRepository itemRepository;

    @Override
    public Item save(Item item) {
        return itemRepository.save(item);
```

```
    }

    @Override
    public void update(Long itemId, ItemUpdateDto updateParam) {
        itemRepository.update(itemId, updateParam);
    }

    @Override
    public Optional<Item> findById(Long id) {
        return itemRepository.findById(id);
    }

    @Override
    public List<Item> findItems(ItemSearchCond cond) {
        return itemRepository.findAll(cond);
    }
}
```

- `ItemServiceV1` 서비스 구현체는 대부분의 기능을 단순히 리포지토리에 위임한다.

컨트롤러 분석

HomeController

```
package hello.itemservice.web;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequiredArgsConstructor
public class HomeController {

    @RequestMapping("/")
    public String home() {
        return "redirect:/items";
    }
```

```
}
```

- 단순히 홈으로 요청이 왔을 때 `items`로 이동하는 컨트롤러이다.

ItemController

```
package hello.itemservice.web;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import hello.itemservice.service.ItemService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import java.util.List;

@Controller
@RequestMapping("/items")
@RequiredArgsConstructor
public class ItemController {

    private final ItemService itemService;

    @GetMapping
    public String items(@ModelAttribute("itemSearch") ItemSearchCond
itemSearch, Model model) {
        List<Item> items = itemService.findItems(itemSearch);
        model.addAttribute("items", items);
        return "items";
    }

    @GetMapping("/{itemId}")
    public String item(@PathVariable long itemId, Model model) {
        Item item = itemService.findById(itemId).get();
        model.addAttribute("item", item);
```

```
        return "item";
    }

    @GetMapping("/add")
    public String addForm() {
        return "addForm";
    }

    @PostMapping("/add")
    public String addItem(@ModelAttribute Item item, RedirectAttributes
redirectAttributes) {
        Item savedItem = itemService.save(item);
        redirectAttributes.addAttribute("itemId", savedItem.getId());
        redirectAttributes.addAttribute("status", true);
        return "redirect:/items/{itemId}";
    }

    @GetMapping("/{itemId}/edit")
    public String editForm(@PathVariable Long itemId, Model model) {
        Item item = itemService.findById(itemId).get();
        model.addAttribute("item", item);
        return "editForm";
    }

    @PostMapping("/{itemId}/edit")
    public String edit(@PathVariable Long itemId, @ModelAttribute ItemUpdateDto
updateParam) {
        itemService.update(itemId, updateParam);
        return "redirect:/items/{itemId}";
    }

}
```

- 상품을 CRUD하는 컨트롤러이다. 자세한 내용은 MVC1편을 참고하자.
- 화면을 출력하기 위한 리소스(`css`, `html`, `templates`)는 MVC1편을 참고하자.

프로젝트 구조 설명2 - 설정

스프링 부트 설정 분석

MemoryConfig

```
package hello.itemservice.config;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.memory.MemoryItemRepository;
import hello.itemservice.service.ItemService;
import hello.itemservice.service.ItemServiceV1;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MemoryConfig {

    @Bean
    public ItemService itemService() {
        return new ItemServiceV1(itemRepository());
    }

    @Bean
    public ItemRepository itemRepository() {
        return new MemoryItemRepository();
    }

}
```

- `ItemServiceV1`, `MemoryItemRepository`를 스프링 빈으로 등록하고 생성자를 통해 의존관계를 주입한다.
- 참고로 여기서는 서비스와 리포지토리는 구현체를 편리하게 변경하기 위해, 이렇게 수동으로 빈을 등록했다.
- 컨트롤러는 컴포넌트 스캔을 사용한다.

TestDataInit

```
package hello.itemservice;
```

```java
import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.event.EventListener;

@Slf4j
@RequiredArgsConstructor
public class TestDataInit {

    private final ItemRepository itemRepository;

    /**
     * 확인용 초기 데이터 추가
     */
    @EventListener(ApplicationReadyEvent.class)
    public void initData() {
        log.info("test data init");
        itemRepository.save(new Item("itemA", 10000, 10));
        itemRepository.save(new Item("itemB", 20000, 20));
    }

}
```

- 애플리케이션을 실행할 때 초기 데이터를 저장한다.
- 리스트에서 데이터가 잘 나오는지 편리하게 확인할 용도로 사용한다.
  - 이 기능이 없으면 서버를 실행할 때 마다 데이터를 입력해야 리스트에 나타난다. (메모리여서 서버를 내리면 데이터가 제거된다.)
- `@EventListener(ApplicationReadyEvent.class)` : 스프링 컨테이너가 완전히 초기화를 다 끝내고, 실행 준비가 되었을 때 발생하는 이벤트이다. 스프링이 이 시점에 해당 애노테이션이 붙은 `initData()` 메서드를 호출해준다.
  - 참고로 이 기능 대신 `@PostConstruct` 를 사용할 경우 AOP 같은 부분이 아직 다 처리되지 않은 시점에 호출될 수 있기 때문에, 간혹 문제가 발생할 수 있다. 예를 들어서 `@Transactional` 과 관련된 AOP가 적용되지 않은 상태로 호출될 수 있다.
  - `@EventListener(ApplicationReadyEvent.class)` 는 AOP를 포함한 스프링 컨테이너가 완전히 초기화 된 이후에 호출되기 때문에 이런 문제가 발생하지 않는다.

ItemServiceApplication

```
package hello.itemservice;

import hello.itemservice.config.*;
import hello.itemservice.repository.ItemRepository;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Profile;


@Import(MemoryConfig.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);
  }

}
```

- `@Import(MemoryConfig.class)` : 앞서 설정한 `MemoryConfig`를 설정 파일로 사용한다.
- `scanBasePackages = "hello.itemservice.web"` : 여기서는 컨트롤러만 컴포넌트 스캔을 사용하고, 나머지는 직접 수동 등록한다. 그래서 컴포넌트 스캔 경로를 `hello.itemservice.web` 하위로 지정했다.
- `@Profile("local")` : 특정 프로필의 경우에만 해당 스프링 빈을 등록한다. 여기서는 `local`이라는 이름의 프로필이 사용되는 경우에만 `testDataInit`이라는 스프링 빈을 등록한다. 이 빈은 앞서 본 것인데, 편의상 초기 데이터를 만들어서 저장하는 빈이다.

프로필

스프링은 로딩 시점에 `application.properties`의 `spring.profiles.active` 속성을 읽어서 프로필로 사용한다.
이 프로필은 로컬(나의 PC), 운영 환경, 테스트 실행 등등 다양한 환경에 따라서 다른 설정을 할 때 사용하는 정보이다.
예를 들어서 로컬PC에서는 로컬 PC에 설치된 데이터베이스에 접근해야 하고, 운영 환경에서는 운영 데이터베이스에 접근해야 한다면 서로 설정 정보가 달라야 한다. 심지어 환경에 따라서 다른 스프링 빈을 등록해야 할 수 도 있다. 프로필을 사용하면 이런 문제를 깔끔하게 해결할 수 있다.

main 프로필

`/src/main/resources` 하위의 `application.properties`

```
spring.profiles.active=local
```

- 이 위치의 `application.properties`는 `/src/main` 하위의 자바 객체를 실행할 때 (주로 `main()`) 동작하는 스프링 설정이다. `spring.profiles.active=local`이라고 하면 스프링은 `local`이라는 프로필로 동작한다. 따라서 직전에 설명한 `@Profile("local")`가 동작하고, `testDataInit`가 스프링 빈으로 등록된다.

실행하면 다음과 같은 로그를 확인할 수 있다.

```
The following 1 profile is active: "local"
```

참고로 프로필을 지정하지 않으면 디폴트(`default`) 프로필이 실행된다.

```
No active profile set, falling back to 1 default profile: "default"
```

test 프로필

`/src/test/resources` 하위의 `application.properties`

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

- 이 위치의 `application.properties`는 `/src/test` 하위의 자바 객체를 실행할 때 동작하는 스프링 설정이다.
- 주로 테스트 케이스를 실행할 때 동작한다.
- `spring.profiles.active=test`로 설정하면 스프링은 `test`라는 프로필로 동작한다. 이 경우 직전에 설명한 `@Profile("local")`는 프로필 정보가 맞지 않아서 동작하지 않는다. 따라서 `testDataInit`

이라는 스프링 빈도 등록되지 않고, 초기 데이터도 추가하지 않는다.

```
The following 1 profile is active: "test"
```

- 프로필 기능을 사용해서 스프링으로 웹 애플리케이션을 로컬(`local`)에서 직접 실행할 때는 `testDataInit` 이 스프링 빈으로 등록된다. 따라서 등록한 초기화 데이터를 편리하게 확인할 수 있다.
- 초기화 데이터 덕분에 편리한 점도 있지만, 테스트 케이스를 실행할 때는 문제가 될 수 있다. 테스트에서 이런 데이터가 들어있다면 오류가 발생할 수 있다. 예를 들어서 데이터를 하나 저장하고 전체 카운트를 확인하는데 1이 아니라 `testDataInit` 때문에 데이터가 2건 추가되어서 3이 되는 것이다.
- 프로필 기능 덕분에 테스트 케이스에서는 `test` 프로필이 실행된다. 따라서 `TestDataInit` 는 스프링 빈으로 추가되지 않고, 따라서 초기 데이터도 추가되지 않는다.

참고

> 프로필에 대한 스프링 부트 공식 메뉴얼은 다음을 참고하자
> https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.profiles

참고

> 스프링 프로필에 대한 더 자세한 내용은 다음 출시 예정인 스프링 부트 강의에서 자세히 다룰 예정이다.

프로젝트 구조 설명3 - 테스트

ItemRepositoryTest

```
package hello.itemservice.domain;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import hello.itemservice.repository.memory.MemoryItemRepository;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
class ItemRepositoryTest {

    @Autowired
    ItemRepository itemRepository;

    @AfterEach
    void afterEach() {
        //MemoryItemRepository 의 경우 제한적으로 사용
        if (itemRepository instanceof MemoryItemRepository) {
            ((MemoryItemRepository) itemRepository).clearStore();
        }
    }

    @Test
    void save() {
        //given
        Item item = new Item("itemA", 10000, 10);

        //when
        Item savedItem = itemRepository.save(item);

        //then
        Item findItem = itemRepository.findById(item.getId()).get();
        assertThat(findItem).isEqualTo(savedItem);
    }

    @Test
    void updateItem() {
        //given
        Item item = new Item("item1", 10000, 10);
        Item savedItem = itemRepository.save(item);
```

```
        Long itemId = savedItem.getId();

        //when
        ItemUpdateDto updateParam = new ItemUpdateDto("item2", 20000, 30);
        itemRepository.update(itemId, updateParam);

        //then
        Item findItem = itemRepository.findById(itemId).get();

assertThat(findItem.getItemName()).isEqualTo(updateParam.getItemName());
        assertThat(findItem.getPrice()).isEqualTo(updateParam.getPrice());

assertThat(findItem.getQuantity()).isEqualTo(updateParam.getQuantity());
    }

    @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);

        //둘 다 없음 검증
        test(null, null, item1, item2, item3);
        test("", null, item1, item2, item3);

        //itemName 검증
        test("itemA", null, item1, item2);
        test("temA", null, item1, item2);
        test("itemB", null, item3);

        //maxPrice 검증
        test(null, 10000, item1);

        //둘 다 있음 검증
```

```
        test("itemA", 10000, item1);
    }

    void test(String itemName, Integer maxPrice, Item... items) {
        List<Item> result = itemRepository.findAll(new ItemSearchCond(itemName,
maxPrice));
        assertThat(result).containsExactly(items);
    }
}
```

- `afterEach` : 테스트는 서로 영향을 주면 안된다. 따라서 각각의 테스트가 끝나고 나면 저장한 데이터를 제거해야 한다. `@AfterEach` 는 각각의 테스트의 실행이 끝나는 시점에 호출된다. 여기서는 메모리 저장소를 완전히 삭제해서 다음 테스트에 영향을 주지 않도록 초기화 한다.
- 인터페이스에는 `clearStore()` 가 없기 때문에 `MemoryItemRepository` 인 경우에만 다운 케스팅을 해서 데이터를 초기화한다. 뒤에서 학습하겠지만, 실제 DB를 사용하는 경우에는 테스트가 끝난 후에 트랜잭션을 롤백해서 데이터를 초기화 할 수 있다.
- `save()`
  - 상품을 하나 저장하고 잘 저장되었는지 검증한다.
- `updateItem()`
  - 상품을 하나 수정하고 잘 수정되었는지 검증한다.
- `findItems()`
  - 상품을 찾는 테스트이다.
    - 상품명과 상품 가격 조건을 다양하게 비교하는 것을 확인할 수 있다.
    - 문자의 경우 `null` 조건도 있지만, 빈 문자(`""`)의 경우에도 잘 동작하는지 검증한다.

> **인터페이스를 테스트하자**
> 여기서는 `MemoryItemRepository` 구현체를 테스트 하는 것이 아니라 `ItemRepository` 인터페이스를 테스트하는 것을 확인할 수 있다. 인터페이스를 대상으로 테스트하면 향후 다른 구현체로 변경되었을 때 해당 구현체가 잘 동작하는지 같은 테스트로 편리하게 검증할 수 있다.

데이터베이스 테이블 생성

이제부터 다양한 데이터 접근 기술들을 활용해서 메모리가 아닌 데이터베이스에 데이터를 보관하는 방법을 알아보자.

먼저 H2 데이터베이스에 접근해서 `item` 테이블을 생성하자.

```
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)
);
```

- `generated by default as identity`
  - `identity` 전략이고 하는데, 기본 키 생성을 데이터베이스에 위임하는 방법이다. MySQL의 Auto Increment와 같은 방법이다.
  - 여기서 PK로 사용되는 `id` 는 개발자가 직접 지정하는 것이 아니라 비워두고 저장하면 된다. 그러면 데이터베이스가 순서대로 증가하는 값을 사용해서 넣어준다.

테이블을 생성했으면, 잘 동작하는지 다음 SQL을 실행하고 조회해보자.

등록 쿼리

```
insert into item(item_name, price, quantity) values ('ItemTest', 10000, 10)
```

조회 쿼리

```
select * from item;
```

- 실행하면 데이터베이스가 생성한 `id` 값을 포함해서 등록한 데이터가 잘 저장되어 있는 것을 확인할 수 있다.

참고 - 권장하는 식별자 선택 전략

데이터베이스 기본 키는 다음 3가지 조건을 모두 만족해야 한다.

1. `null` 값은 허용하지 않는다.
2. 유일해야 한다.
3. 변해선 안 된다.

테이블의 기본 키를 선택하는 전략은 크게 2가지가 있다.

- 자연 키(natural key)
  - 비즈니스에 의미가 있는 키
  - 예: 주민등록번호, 이메일, 전화번호
- 대리 키(surrogate key)
  - 비즈니스와 관련 없는 임의로 만들어진 키, 대체 키로도 불린다.
  - 예: 오라클 시퀀스, auto_increment, identity, 키생성 테이블 사용

자연 키보다는 대리 키를 권장한다

자연 키와 대리 키는 일장 일단이 있지만 될 수 있으면 대리 키의 사용을 권장한다. 예를 들어 자연 키인 전화번호를 기본 키로 선택한다면 그 번호가 유일할 수는 있지만, 전화번호가 없을 수도 있고 전화번호가 변경될 수도 있다. 따라서 기본 키로 적당하지 않다. 문제는 주민등록번 호처럼 그럴듯하게 보이는 값이다. 이 값은 `null`이 아니고 유일하며 변하지 않는다는 3가지 조건을 모두 만족하는 것 같다. 하지만 현실과 비즈니스 규칙은 생각보다 쉽게 변한다. 주민등록번호 조차도 여러 가지 이유로 변경될 수 있다.

비즈니스 환경은 언젠가 변한다

나의 경험을 하나 이야기하겠다. 레거시 시스템을 유지보수할 일이 있었는데, 분석해보니 회원 테이블에 주민등록번호가 기본 키로 잡혀 있었다. 회원과 관련된 수많은 테이블에서 조인을 위 해 주민등록번호를 외래 키로 가지고 있었고 심지어 자식 테이블의 자식 테이블까지 주민등록 번호가 내려가 있었다. 문제는 정부 정책이 변경되면서 법적으로 주민등록번호를 저장할 수 없게 되면서 발생했다. 결국 데이터베이스 테이블은 물론이고 수많은 애플리케이션 로직을 수정 했다. 만약 데이터베이스를 처음 설계할 때부터 자연 키인 주민등록번호 대신에 비즈니스와 관련 없는 대리 키를 사용했다면 수정할 부분이 많지는 않았을 것이다.

기본 키의 조건을 현재는 물론이고 미래까지 충족하는 자연 키를 찾기는 쉽지 않다. 대리 키는 비즈니스와 무관한 임의의 값이므로 요구사항이 변경되어도 기본 키가 변경되는 일은 드물다. 대리 키를 기본 키로 사용하되 주민등록번호나 이메일처럼 자연 키의 후보가 되는 컬럼들은 필요에 따라 유니크 인덱스를 설정해서 사용하는 것을 권장한다.

참고로 JPA는 모든 엔티티에 일관된 방식으로 대리 키 사용을 권장한다

비즈니스 요구사항은 계속해서 변하는데 테이블은 한 번 정의하면 변경하기 어렵다. 그런면에 서 외부 풍파에 쉽게 흔들리지 않는 대리 키가 일반적으로 좋은 선택이라 생각한다.

정리