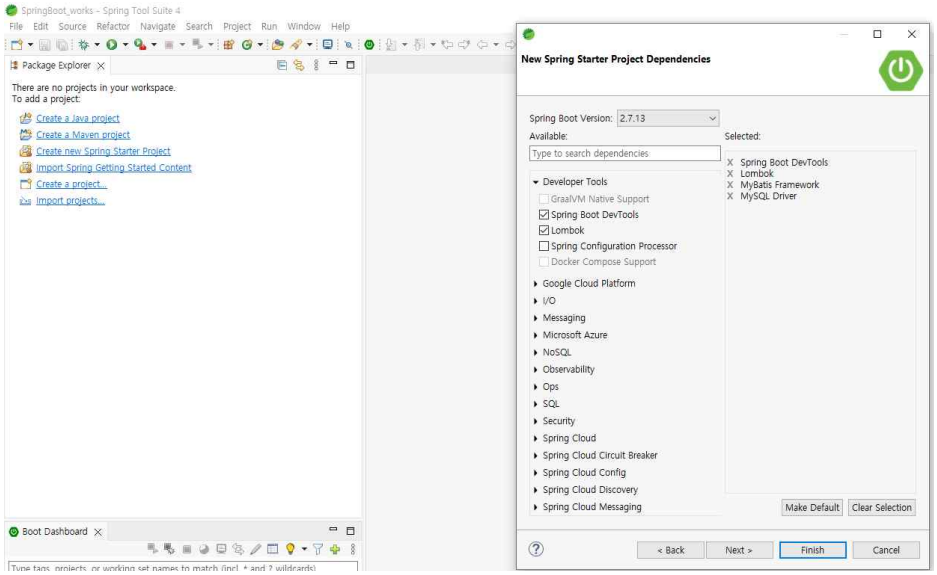
Spring Framework STS4

<Spring Boot>
Spring 4 버전 다운로드 - 라이브러리 필요한거 추가하면 됨(Maven Repository 가서 일일이 소스 붙일 필요 없음)




```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
```

시작이 됨 표시(SPRING 글자표시)

DB에 관한 것 받고 설정을 해줘야 함.
web.xml없이도 자동으로 설정이 됨.
spring boot에서는 java로 환경설정을 함.

application.properties - 환경설정
server.port=7777

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
spring.datasource.dbcp2.username=root
spring.datasource.dbcp2.password=root

<Spring Boot - STS4>

Lombok, MyBatis Framework, MySQL Driver, Spring Boot, Spring Boot DevTools, Spring Web 선택

Run As - Spring Boot

<JSP 사용 환경 설정>
server.port=7777

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
spring.datasource.dbcp2.username=root
spring.datasource.dbcp2.password=root

<!-- JSP 파일 설정-->
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp

spring.devtools.livereload.enabled=true

서버가동 후 지구본 모양 클릭하면 이와 같이 뜸
Whitelabel Error Page
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Jul 04 09:12:49 KST 2023
There was an unexpected error (type=Not Found, status=404).
No message available

<Jasper>
maven - tomcat-embed-jasper 10.0.22
버전은 지우고 pom.xml에서 사용

maven - JavaServer Pages(TM) Standard Tag Library 1.2버전
Eclipse Marketplace - Eclipse Enterprise Java and Web Developer Tools 설치
EUC-KR -> UTF-8로 인코딩 설정

Legacy에서 / 인식이 된 것
xml에서는 모두 설정했음.
부트에서는 직접적으로 모든 것들 해줌.

<MyBatis 환경설정>
#XML Location
mybatis.mapper-locations=classpath:mappers/*.xml

Spring Boot Security

<Security 설정>
<!-- https://mvnrepository.com/artifact/org.springframework.security/spring-security-taglibs
-->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>

```
</dependency>

server.port=8282

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url
=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password=root

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp

mybatis.type-aliases-package=com.example.demo02.dto

spring.devtools.livereload.enabled=true
```

-> 패키지 찾는 경로 지정

<JPA>
JPA(Java Persistence API)는 Java 언어를 위한 ORM(Object-Relational Mapping) 기술입니다. ORM은 객체와 관계형 데이터베이스 간의 매핑을 자동으로 처리하여 개발자가 SQL 쿼리를 직접 작성하지 않고도 데이터베이스를 조작할 수 있도록 도와줍니다.

JPA는 데이터베이스와의 상호작용을 위한 일종의 인터페이스로서, ORM 프레임워크들이 JPA 인터페이스를 구현하여 사용합니다. 대표적인 JPA 구현체로는 Hibernate, EclipseLink, OpenJPA 등이 있습니다. Spring Framework에서는 Hibernate를 기반으로 한 Spring Data JPA를 제공하여 개발자들이 편리하게 JPA를 사용할 수 있도록 지원하고 있습니다.

JPA를 사용하면 Java 클래스와 데이터베이스 테이블 간의 매핑을 어노테이션을 통해 설정하고, 객체 지향적인 방식으로 데이터를 다룰 수 있습니다. JPA를 사용하면 CRUD(Create, Read, Update, Delete) 작업을 위한 메서드를 간편하게 작성하고, 데이터베이스와의 일관성을 유지할 수 있습니다. 또한 JPA는 객체 그래프 탐색, 지연 로딩, 트랜잭션 관리 등 다양한 기능을 제공합니다.

JPA를 사용하면 개발자는 데이터베이스와의 상호작용에 대한 복잡성을 줄이고, 객체 지향적인 프로그래밍으로 개발을 진행할 수 있습니다. 또한 데이터베이스 종속성을 줄여 이식성을 향상시킬 수 있습니다.

<JPA 환경설정>
Starter Project - JPA추가
리소스에서 아래와 같이 추가

```
server.port=8383

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url
=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
```

```
spring.datasource.username = root
spring.datasource.password=root

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp


spring.devtools.livereload.enabled=true

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

MyBatis는 사용을 안하므로 MyBatis 부분은 삭제

패키지 controller, model, repository, service 생성

<HomeController>

```
package com.example.demo03.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

import com.example.demo03.model.JpaMember;
import com.example.demo03.service.MemberService;

@Controller
public class HomeController {
	@Autowired
	private MemberService memberService;

	@GetMapping("/")
	public String home() {
		return "home";
	}
	// Insert Form
	@GetMapping("/join")
	public String join() {
		return "join";
	}

	//Insert
	@PostMapping("/join")
	public String join(JpaMember member) {
		memberService.insert(member);
		return "redirect:list";
	}
```

```
	@GetMapping("list")
	public String list(Model model) {
		model.addAttribute("lists", memberService.list());
		return "list";
	}
}
```

<JPA Member 생성>

```
package com.example.demo03.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

import lombok.Data;

@Data
@Entity
public class JpaMember {
	@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private String name;
	private String password;
	private String email;
	private String memo;
	@Column(name = "address")
	private String addr;
}
```

<@Entity>

@Entity는 JPA(Java Persistence API)에서 사용되는 어노테이션입니다. 이 어노테이션은 클래스를 엔티티(Entity)로 지정하며, 해당 클래스가 데이터베이스의 테이블과 매핑되는 것을 나타냅니다.

@Entity 어노테이션이 지정된 클래스는 JPA에서 관리하는 객체(Entity)로 간주됩니다. 이 클래스의 인스턴스는 데이터베이스의 테이블과 일대일로 매핑되어 데이터베이스와 상호작용할 수 있게 됩니다. 즉, 엔티티 클래스는 데이터베이스의 테이블과 필드(column) 간의 매핑을 정의하고, JPA를 통해 해당 데이터를 조작하고 검색할 수 있습니다.

@Entity 어노테이션은 주로 클래스 레벨에 적용되며, JPA가 엔티티 클래스를 인식하고 관리할 수 있도록 합니다. 엔티티 클래스에는 데이터베이스의 테이블과 매핑되는 필드들과 관련된 많은 어노테이션들을 사용할 수 있습니다. 예를 들어, @Id 어노테이션은 주요 식별자(primary key) 필드를 지정하고, @Column 어노테이션은 열(column)에 대한 매핑을 지정할 수 있습니다.

@Entity 어노테이션을 사용하여 클래스를 엔티티로 지정하면, JPA는 해당 클래스를 데이터베이스 스키마에 맞게 매핑하여 필요한 SQL 쿼리를 자동으로 생성하고 실행할 수 있습니다. 이를 통해 객체 지향적인 방식으로 데이터베이스를 다룰 수 있으며, 데이터베이스 스키마 변경 시에도 쉽게 대응할 수 있습니다.

<Repository>

```
package com.example.demo03.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo03.model.JpaMember;

public interface MemberRepository extends JpaRepository<JpaMember, Long> {

}
```

앞에서는 @Repository를 사용하여 지정했음 JPA에서는 이와 같이 생성함.

<MemberService>

```
package com.example.demo03.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo03.model.JpaMember;
import com.example.demo03.repository.MemberRepository;

import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class MemberService {
//	@Autowired
//	private MemberRepository memberRepository;
	private final MemberRepository memberRepository;
	// 추가
	public void insert(JpaMember member) {
		memberRepository.save(member);
	}
	//전체보기
	public List<JpaMember> list() {
		return memberRepository.findAll();
	}
}
```

@Service: 스프링의 서비스(Service) 계층을 나타내는 어노테이션입니다. 해당 클래스가 비즈니스 로직을 처리하는 서비스 클래스임을 나타냅니다.

@RequiredArgsConstructor: Lombok 어노테이션으로, 클래스의 생성자를 자동으로 생성해줍니다. 주입받아야 하는 필드를 final 키워드로 선언하고, 해당 어노테이션을 붙여주면 해당 필드를 인자로 받는 생성자가 자동으로 생성됩니다.

MemberService 클래스는 MemberRepository 인터페이스를 주입받아 사용합니다.
MemberRepository는 데이터베이스와의 상호작용을 담당하는 인터페이스입니다.

insert 메서드는 회원 정보를 저장하는 기능을 구현합니다. memberRepository.save(member)를 호출하여 회원 정보를 데이터베이스에 저장합니다.

list 메서드는 모든 회원 정보를 조회하는 기능을 구현합니다. memberRepository.findAll()을 호출하여 모든 회원 정보를 리스트 형태로 반환합니다.

-> jpa_member의 이름으로 CRUD를 자동적으로 할 수 있는 SQL Query Table을 만들어준다.

<home.jsp 구성>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>HOME</h1>
<a href="/join">insert</a><br/>
<a href="/list">list</a>
</body>
</html>
```

<list.jsp 구성>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>MEMBER LIST</h1>
        <c:forEach items="${lists }" var="member">
        아이디 : ${member.id }<br/>
        이름 : ${member.name }<br/>
        이메일 : ${member.email }<br/>
        주소 : ${member.addr }<br/>
        메모 : ${member.memo }<br/><br/>
        </c:forEach>

</body>
</html>
```

<join.jsp 구성>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>JOIN</h1>
<form action="/join" method="post">
Name : <input type="text" name="name"/><br/>
Password : <input type="password" name="password"/><br/>
Email : <input type="text" name="email"/><br/>
Addr : <input type="text" name="addr" size="30"/><br/>
Memo : <textarea rows="5" cols="50" name="memo"></textarea><br/>
<button type="submit">Submit</button>
</form>
</body>
</html>
```

JPA와 Entity, 영속성 컨텍스트, E-R 모델링

<JPA>
JPA(Java Persistence API)는 Java 언어를 위한 ORM(Object-Relational Mapping) 기술입니다. ORM은 객체와 관계형 데이터베이스 간의 매핑을 자동으로 처리하여 개발자가 SQL 쿼리를 직접 작성하지 않고도 데이터베이스를 조작할 수 있도록 도와줍니다.

JPA는 데이터베이스와의 상호작용을 위한 일종의 인터페이스로서, ORM 프레임워크들이 JPA 인터페이스를 구현하여 사용합니다. 대표적인 JPA 구현체로는 Hibernate, EclipseLink, OpenJPA 등이 있습니다. Spring Framework에서는 Hibernate를 기반으로 한 Spring Data JPA를 제공하여 개발자들이 편리하게 JPA를 사용할 수 있도록 지원하고 있습니다.

JPA를 사용하면 Java 클래스와 데이터베이스 테이블 간의 매핑을 어노테이션을 통해 설정하고, 객체 지향적인 방식으로 데이터를 다룰 수 있습니다. JPA를 사용하면 CRUD(Create, Read, Update, Delete) 작업을 위한 메서드를 간편하게 작성하고, 데이터베이스와의 일관성을 유지할 수 있습니다. 또한 JPA는 객체 그래프 탐색, 지연 로딩, 트랜잭션 관리 등 다양한 기능을 제공합니다.

JPA를 사용하면 개발자는 데이터베이스와의 상호작용에 대한 복잡성을 줄이고, 객체 지향적인 프로그래밍으로 개발을 진행할 수 있습니다. 또한 데이터베이스 종속성을 줄여 이식성을 향상시킬 수 있습니다.

<JPA 환경설정>
Starter Project - JPA추가
리소스에서 아래와 같이 추가
server.port=8383

```
#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url
=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password=root

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp


spring.devtools.livereload.enabled=true

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

MyBatis는 사용을 안하므로 MyBatis 부분은 삭제

패키지 controller, model, repository, service 생성

<HomeController>

```
package com.example.demo03.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

import com.example.demo03.model.JpaMember;
import com.example.demo03.service.MemberService;

@Controller
public class HomeController {
	@Autowired
	private MemberService memberService;

	@GetMapping("/")
	public String home() {
		return "home";
	}
	// Insert Form
	@GetMapping("/join")
	public String join() {
		return "join";
	}
```

```
        //Insert
        @PostMapping("/join")
        public String join(JpaMember member) {
                memberService.insert(member);
                return "redirect:list";
        }
        @GetMapping("list")
        public String list(Model model) {
                model.addAttribute("lists", memberService.list());
                return "list";
        }
}
```

<JPA Member 생성>

```
package com.example.demo03.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

import lombok.Data;

@Data
@Entity
public class JpaMember {
        @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String name;
        private String password;
        private String email;
        private String memo;
        @Column(name = "address")
        private String addr;
}
```

<@Entity>

@Entity는 JPA(Java Persistence API)에서 사용되는 어노테이션입니다. 이 어노테이션은 클래스를 엔티티(Entity)로 지정하며, 해당 클래스가 데이터베이스의 테이블과 매핑되는 것을 나타냅니다.

@Entity 어노테이션이 지정된 클래스는 JPA에서 관리하는 객체(Entity)로 간주됩니다. 이 클래스의 인스턴스는 데이터베이스의 테이블과 일대일로 매핑되어 데이터베이스와 상호작용할 수 있게 됩니다. 즉, 엔티티 클래스는 데이터베이스의 테이블과 필드(column) 간의 매핑을 정의하고, JPA를 통해 해당 데이터를 조작하고 검색할 수 있습니다.

@Entity 어노테이션은 주로 클래스 레벨에 적용되며, JPA가 엔티티 클래스를 인식하고 관리할 수 있도록 합니다. 엔티티 클래스에는 데이터베이스의 테이블과 매핑되는 필드들과 관련된 많은 어노테이션들을 사용할 수 있습니다. 예를 들어, @Id 어노테이션은 주요 식별자(primary key) 필드를 지정하고, @Column 어노테이션은 열(column)에 대한 매핑을 지정할 수 있습니다.

@Entity 어노테이션을 사용하여 클래스를 엔티티로 지정하면, JPA는 해당 클래스를 데이터베이스 스키마에 맞게 매핑하여 필요한 SQL 쿼리를 자동으로 생성하고 실행할 수 있습니다. 이를 통해 객체 지향적인 방식으로 데이터베이스를 다룰 수 있으며, 데이터베이스 스키마 변경 시에도 쉽게 대응할 수 있습니다.

<Repository>

```
package com.example.demo03.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo03.model.JpaMember;

public interface MemberRepository extends JpaRepository<JpaMember, Long> {

}
```

앞에서는 @Repository를 사용하여 지정했음 JPA에서는 이와 같이 생성함.

<MemberService>

```
package com.example.demo03.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo03.model.JpaMember;
import com.example.demo03.repository.MemberRepository;

import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class MemberService {
//	@Autowired
//	private MemberRepository memberRepository;
	private final MemberRepository memberRepository;
```

스프링에서는 @Autowired 보다는 생성자를 사용하여 만드는 것을 권고하는편임

```
	// 추가
	public void insert(JpaMember member) {
		memberRepository.save(member);
	}
	// 전체보기
	public List<JpaMember> list() {
		return memberRepository.findAll();
	}
}
```

@Service: 스프링의 서비스(Service) 계층을 나타내는 어노테이션입니다. 해당 클래스가 비즈니스 로직을 처리하는 서비스 클래스임을 나타냅니다.

@RequiredArgsConstructor: Lombok 어노테이션으로, 클래스의 생성자를 자동으로 생성해줍니다. 주입받아야 하는 필드를 final 키워드로 선언하고, 해당 어노테이션을 붙여주면 해당 필드를 인자로 받는 생성자가 자동으로 생성됩니다.

MemberService 클래스는 MemberRepository 인터페이스를 주입받아 사용합니다.
MemberRepository는 데이터베이스와의 상호작용을 담당하는 인터페이스입니다.

insert 메서드는 회원 정보를 저장하는 기능을 구현합니다. memberRepository.save(member)를 호출하여 회원 정보를 데이터베이스에 저장합니다.

list 메서드는 모든 회원 정보를 조회하는 기능을 구현합니다. memberRepository.findAll()을 호출하여 모든 회원 정보를 리스트 형태로 반환합니다.

-> jpa_member의 이름으로 CRUD를 자동적으로 할 수 있는 SQL Query Table을 만들어준다.

<home.jsp 구성>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>HOME</h1>
<a href="/join">insert</a><br/>
<a href="/list">list</a>
</body>
</html>
```

<list.jsp 구성>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>MEMBER LIST</h1>
        <c:forEach items="${lists }" var="member">
        아이디 : ${member.id }<br/>
        이름 : ${member.name }<br/>
```

```
        이메일 : ${member.email }<br/>
        주소 : ${member.addr }<br/>
        메모 : ${member.memo }<br/><br/>
        </c:forEach>

</body>
</html>
```

<join.jsp 구성>

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>JOIN</h1>
<form action="/join" method="post">
Name : <input type="text" name="name"/><br/>
Password : <input type="password" name="password"/><br/>
Email : <input type="text" name="email"/><br/>
Addr : <input type="text" name="addr" size="30"/><br/>
Memo : <textarea rows="5" cols="50" name="memo"></textarea><br/>
<button type="submit">Submit</button>
</form>
</body>
</html>
```

<Optional 속성>
Optional은 자바 8부터 도입된 클래스로, 값이 존재할 수도 있고 존재하지 않을 수도 있는 객체를 감싸고 있는 래퍼 클래스입니다. Optional은 주로 null을 대체하고 예외 처리를 간편하게 할 수 있도록 도와줍니다.

Optional의 주요 특징은 다음과 같습니다:

Optional 객체는 값이 있을 수도 있고 없을 수도 있습니다. 값이 있는 경우 Optional 객체는 해당 값을 감싸고, 값이 없는 경우 Optional 객체는 비어있는 상태입니다.
Optional 객체는 null을 감싸는 것보다 안전하고 명시적인 방식입니다. null을 다룰 때 발생하는 NullPointerException을 방지할 수 있습니다.
Optional 객체를 사용하여 예외 처리를 간결하게 할 수 있습니다. 값이 존재하지 않을 경우 예외를 던지거나 대체 값을 제공하는 등의 처리가 가능합니다.

<영속성 컨텍스트>
flush => DB
Transaction 처리하면 Flush 처리가 된다.
영속성 컨텍스트(Persistence Context)는 JPA(Java Persistence API)에서 엔티티(Entity)를 관리하는 논리적인 영역입니다. 영속성 컨텍스트는 엔티티의 상태를 추적하고, 데이터베이스와의

통신을 최소화하여 성능을 향상 시키는 기능을 제공합니다.

<@JoinColumn, @OneToOne>
@JoinColumn은 JPA에서 사용되는 어노테이션으로, 연관관계의 주인(Owner) 쪽 엔티티에서 사용됩니다. 이 어노테이션은 외래 키(Foreign Key) 컬럼과 매핑하기 위해 사용됩니다.
@JoinColumn은 다음과 같은 속성을 가질 수 있습니다:

name: 외래 키 컬럼의 이름을 지정합니다. 기본값은 "참조하는 테이블의 기본 키 컬럼명_참조하는 테이블의 기본 키 컬럼명" 형식입니다.
referencedColumnName: 참조하는 테이블의 기본 키 컬럼과 매핑할 참조하는 테이블의 컬럼 이름을 지정합니다. 기본값은 참조하는 테이블의 기본 키 컬럼과 동일한 이름입니다.
nullable: 외래 키 컬럼이 NULL 값 허용 여부를 지정합니다. 기본값은 true입니다.
unique: 외래 키 컬럼에 유니크 제약 조건을 설정할지 여부를 지정합니다. 기본값은 false입니다.
insertable: 외래 키 컬럼이 INSERT 문에 포함되어야 하는지 여부를 지정합니다. 기본값은 true입니다.
updatable: 외래 키 컬럼이 UPDATE 문에 포함되어야 하는지 여부를 지정합니다. 기본값은 true입니다.
@OneToOne은 JPA에서 사용되는 어노테이션으로, 일대일(One-to-One) 관계를 매핑하기 위해 사용됩니다. 이 어노테이션은 두 개의 엔티티 간에 일대일 관계가 있는 경우 사용됩니다.
@OneToOne은 다음과 같은 속성을 가질 수 있습니다:

targetEntity: 대상 엔티티의 클래스를 지정합니다. 기본값은 필드 또는 프로퍼티의 타입으로 추론됩니다.
cascade: 연관된 엔티티에 대해 어떤 작업(예: 저장, 업데이트, 삭제)을 수행할 때 연쇄적으로 적용할 작업을 지정합니다.
fetch: 연관된 엔티티를 조회할 때 어떻게 가져올지 지정합니다(지연 로딩, 즉시 로딩 등).
mappedBy: 양방향 관계에서 반대쪽 엔티티에서 연관관계의 주인을 지정합니다.
optional: 연관된 엔티티가 선택적인지 여부를 지정합니다. 기본값은 true입니다.
@OneToOne 어노테이션을 사용하여@JoinColumn은 JPA에서 사용되는 어노테이션으로, 연관관계 매핑 시 외래 키(Foreign Key) 컬럼을 지정하는데 사용됩니다. 주로 일대일(One-to-One) 또는 다대일(Many-to-One) 관계에서 사용됩니다. @JoinColumn은 다음과 같은 속성을 가질 수 있습니다:

name: 외래 키 컬럼의 이름을 지정합니다. 기본값은 "참조하는 테이블의 기본 키 컬럼명_참조하는 테이블의 기본 키 컬럼명" 형식입니다.
referencedColumnName: 참조하는 테이블의 기본 키 컬럼과 매핑할 참조하는 테이블의 컬럼 이름을 지정합니다. 기본값은 참조하는 테이블의 기본 키 컬럼과 동일한 이름입니다.
nullable: 외래 키 컬럼이 NULL 값을 허용하는지 여부를 지정합니다. 기본값은 true입니다.
unique: 외래 키 컬럼에 유니크 제약 조건을 설정할지 여부를 지정합니다. 기본값은 false입니다.
insertable: 외래 키 컬럼이 INSERT 문에 포함되어야 하는지 여부를 지정합니다. 기본값은 true입니다.
updatable: 외래 키 컬럼이 UPDATE 문에 포함되어야 하는지 여부를 지정합니다. 기본값은 true입니다.
@OneToOne은 JPA에서 사용되는 어노테이션으로, 일대일(One-to-One) 관계를 매핑하기 위해 사용됩니다. @OneToOne 어노테이션은 두 개의 엔티티 간에 일대일 관계가 있는 경우 사용됩니다. 이 어노테이션은 다음과 같은 속성을 가질 수 있습니다:

targetEntity: 대상 엔티티의 클래스를 지정합니다. 기본값은 필드 또는 프로퍼티의 타입으로 추론됩니다.

cascade: 연관된 엔티티에 대해 어떤 작업(예: 저장, 업데이트, 삭제)을 수행할 때 연쇄적으로 적용할 작업을 지정합니다.
fetch: 연관된 엔티티를 조회할 때 어떻게 가져올지 지정합니다(지연 로딩, 즉시 로딩 등).
mappedBy: 양방향 관계에서 반대쪽 엔티티에서 연관관계의 주인을 지정합니다.
optional: 연관된 엔티티가 선택적인지 여부를 지정합니다. 기본값은 true입니다.
이러한@Entity는 JPA(Java Persistence API)에서 사용되는 어노테이션으로, 해당 클래스가 엔티티(Entity)임을 표시하는 역할을 합니다. 엔티티는 데이터베이스의 테이블과 매핑되는 개념이며, @Entity 어노테이션을 사용하여 해당 클래스를 엔티티로 지정합니다.

@Entity 어노테이션은 다음과 같은 주요 속성을 가지고 있습니다:

name: 엔티티의 이름을 지정합니다. 기본적으로는 클래스의 이름이 사용됩니다.
catalog: 엔티티가 속하는 데이터베이스 카탈로그의 이름을 지정합니다.
schema: 엔티티가 속하는 데이터베이스 스키마의 이름을 지정합니다.
uniqueConstraints: 엔티티에 대한 고유 제약 조건을 정의합니다.
indexes: 엔티티에 대한 인덱스를 정의합니다.
@Entity 어노테이션이 지정된 클래스는 JPA의 관리 대상이 되며, 해당 클래스의 인스턴스는 데이터베이스의 테이블과 매핑되어 영속성 컨텍스트에서 관리됩니다. 이를 통해 객체와 데이터베이스 간의 매핑을 편리하게 처리할 수 있습니다.

<@Embedded, @Embeddable>
@Embedded와 @Embeddable은 JPA(Java Persistence API)에서 사용되는 어노테이션으로, 객체를 데이터베이스에 임베딩(포함)하기 위해 사용됩니다.

@Embeddable은 엔티티 클래스 내에서 재사용 가능한 임베디드 타입을 정의하기 위해 사용됩니다. @Embeddable 어노테이션이 적용된 클래스는 해당 엔티티 클래스의 속성으로 사용될 수 있습니다. 즉, 엔티티 클래스 내에서 다른 객체로 포함될 수 있는 재사용 가능한 구성요소를 정의합니다.

@Embedded는 엔티티 클래스 내의 속성으로 임베디드 타입을 사용하기 위해 지정됩니다. @Embedded 어노테이션이 적용된 속성은 해당 엔티티의 테이블에서 별도의 컬럼으로 저장되지 않고, 임베디드 타입의 속성들이 함께 속성을 구성하게 됩니다.

<E-R 모델링(Entity-Relationship Modeling)>
Entity-Relationship Modeling은 데이터베이스 설계를 위해 사용되는 개념적 모델링 기법입니다. 이 모델링 기법은 현실 세계의 엔티티(Entity)와 그들 간의 관계(Relationship)를 표현하여 데이터베이스의 구조를 시각적으로 설명합니다.

Entity-Relationship Modeling은 데이터베이스 설계자와 비즈니스 사용자 간의 의사 소통을 원활하게 하고, 데이터베이스의 요구사항을 명확히 파악하여 데이터베이스 스키마를 정의하는 데 도움을 줍니다.

Entity는 현실 세계에서 독립적으로 존재하는 개체를 나타내며, 데이터베이스 테이블의 레코드에 해당합니다. 각 Entity는 속성(Attribute)을 가지며, 속성은 Entity의 특징이나 속성을 나타냅니다. 예를 들어, "고객" Entity는 "이름", "나이", "주소"와 같은 속성을 가질 수 있습니다.

Relationship은 Entity 간의 관계를 나타냅니다. 관계는 Entity 간의 연결이며, 일대일(One-to-One), 일대다(One-to-Many), 다대다(Many-to-Many)와 같은 다양한 유형의 관계가 있습니다. 예를 들어, "주문" Entity와 "제품" Entity는 일대다 관계일 수 있으며, 한 주문에는 여러 개의 제품이 속할 수 있습니다.

Entity-Relationship Modeling은 Entity와 Relationship을 다이어그램 형태로 그려서 데이터베이스 구조를 시각화합니다. 이 다이어그램을 통해 데이터베이스의 구조, 엔티티 간의 관계, 속성 및 제약 조건을 이해할 수 있습니다. 이를 기반으로 데이터베이스 테이블을 설계하고, 데이터베이스 시스템을 구축할 수 있습니다.

<yaml config 파일(application.yml)로 구현>

```yaml
server:
  port: 8383
spring:
  datasource:
    dbcp2:
      driver-class-name: com.mysql.cj.jdbc.Driver
    password: root
    url:
jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEnc
oding=UTF-8
    username: root
  devtools:
    livereload:
      enabled: true
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
  mvc:
    view:
      prefix: /WEB-INF/views/
      suffix: .jsp
```

<기존 세팅>

```
server.port=8383

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url
=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEn
coding=UTF-8
spring.datasource.username = root
spring.datasource.password=root

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp

spring.devtools.livereload.enabled=true

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

같은 의미를 지닌다.

<Spring Boot JPA를 활용한 시큐리티 설정>
<SecurityConfig 클래스 생성>

```
package com.example.demo04.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {
	@Bean
	public BCryptPasswordEncoder encodePwd() {
		return new BCryptPasswordEncoder();
	}
	@Bean
	public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
		http.csrf().disable()
			.authorizeHttpRequests()
			.antMatchers("/user/*").authenticated()
			.anyRequest()
			.permitAll()
		.and()
			.formLogin()
			.loginPage("/login")
			// .loginProcessingUrl("/loginPro")
			.defaultSuccessUrl("/")
		.and()
			.logout()
			.logoutUrl("/logout")
			.logoutSuccessUrl("/")
			.invalidateHttpSession(true);

		return http.build();
	}
}
```

<model 생성>
<user 클래스 생성>

```
package com.example.demo04.model;

import javax.persistence.Column;

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

```
import lombok.Getter;
import lombok.Setter;

@Getter @Setter
@Table(name = "tbl_user4")
@Entity
public class User {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	@Column(nullable = false)
	private String username;
	private String password;
	private String email;
	private String role;

}
```

<Board 클래스 생성>

```
package com.example.demo04.model;

import java.util.Date;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.PrePersist;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

import org.hibernate.annotations.CreationTimestamp;

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
@Table
@Entity(name = "tbl_board4")
public class Board {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long num;
```

```
	private String title;
	// private String writer;
	private String content;
	@CreationTimestamp
	@Temporal(TemporalType.TIMESTAMP)
	@Column(name="regdate")
	private Date regdate;
	private Long hitcount;
	private Long replycnt;

	////// ???
	@OneToMany(mappedBy = "board")
	private List<Comment> comments;
	@ManyToOne
	@JoinColumn(name="user_id")
	private User user;

	@PrePersist
	public void prePersist() {
		this.hitcount=this.hitcount==null?0:this.hitcount;
		this.replycnt=this.replycnt==null?0:this.replycnt;
	}
}
```

<Comment 클래스 생성>

```
package com.example.demo04.model;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

import org.hibernate.annotations.CreationTimestamp;

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
@Table
@Entity(name = "tbl_comment4")
public class Comment {
	@Id
```

```
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long cnum;
	private String content;
	@CreationTimestamp
	@Temporal(TemporalType.TIMESTAMP)
	@Column(name = "regdate")
	private Date regdate;
	// bnum
	@ManyToOne
	@JoinColumn(name="bnum")
	private Board board;
	// userid
	@ManyToOne
	@JoinColumn(name="user_id")
	private User user;
}
```

<Config 파일 환경 설정>

```
server.port=8844

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url
=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password=root

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp

spring.devtools.livereload.enabled=true

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

<pom.xml>

```
<dependency>
		<groupId>org.apache.tomcat.embed</groupId>
		<artifactId>tomcat-embed-jasper</artifactId>
	</dependency>
	<dependency>
		<groupId>javax.servlet</groupId>
		<artifactId>jstl</artifactId>
	</dependency>

<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-taglibs</artifactId>
```

```
		</dependency>
```

추가 해주기

<Spring Boot JPA를 활용한 시큐리티 설정>
<SecurityConfig 클래스 생성>

```
package com.example.demo04.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {
	@Bean
	public BCryptPasswordEncoder encodePwd() {
		return new BCryptPasswordEncoder();
	}
	@Bean
	public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
		http.csrf().disable()
			.authorizeHttpRequests()
			.antMatchers("/user/*").authenticated()
			.anyRequest()
			.permitAll()
		.and()
			.formLogin()
			.loginPage("/login")
			// .loginProcessingUrl("/loginPro")
			.defaultSuccessUrl("/")
		.and()
			.logout()
			.logoutUrl("/logout")
			.logoutSuccessUrl("/")
			.invalidateHttpSession(true);

		return http.build();
	}
}
```

<model 생성>
<user 클래스 생성>

```
package com.example.demo04.model;

import javax.persistence.Column;
```

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

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
@Table(name = "tbl_user4")
@Entity
public class User {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	@Column(nullable = false)
	private String username;
	private String password;
	private String email;
	private String role;

}
```

<Board 클래스 생성>

```
package com.example.demo04.model;

import java.util.Date;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.PrePersist;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

import org.hibernate.annotations.CreationTimestamp;

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
```

```
@Table
@Entity(name = "tbl_board4")
public class Board {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long num;
	private String title;
	// private String writer;
	private String content;
	@CreationTimestamp
	@Temporal(TemporalType.TIMESTAMP)
	@Column(name="regdate")
	private Date regdate;
	private Long hitcount;
	private Long replycnt;

	////// ???
	@OneToMany(mappedBy = "board")
	private List<Comment> comments;
	@ManyToOne
	@JoinColumn(name="user_id")
	private User user;

	@PrePersist
	public void prePersist() {
		this.hitcount=this.hitcount==null?0:this.hitcount;
		this.replycnt=this.replycnt==null?0:this.replycnt;
	}
}
```

<Comment 클래스 생성>

```
package com.example.demo04.model;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

import org.hibernate.annotations.CreationTimestamp;

import lombok.Getter;
import lombok.Setter;
```

```
@Getter @Setter
@Table
@Entity(name = "tbl_comment4")
public class Comment {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long cnum;
	private String content;
	@CreationTimestamp
	@Temporal(TemporalType.TIMESTAMP)
	@Column(name = "regdate")
	private Date regdate;
	// bnum
	@ManyToOne
	@JoinColumn(name="bnum")
	private Board board;
	// userid
	@ManyToOne
	@JoinColumn(name="user_id")
	private User user;
}
```

<Config 파일 환경 설정>

```
server.port=8844

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url
=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password=root

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp

spring.devtools.livereload.enabled=true

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

<pom.xml>

```
<dependency>
			<groupId>org.apache.tomcat.embed</groupId>
			<artifactId>tomcat-embed-jasper</artifactId>
		</dependency>
		<dependency>
			<groupId>javax.servlet</groupId>
```

```
            <artifactId>jstl</artifactId>
        </dependency>

<dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-taglibs</artifactId>
        </dependency>
```

추가 해주기

나머지 부분은 소스코드 파일 참조

영속성 개념, 영속성 활용하기, JPQL 활용

<영속성>
영속성(Persistence)은 데이터를 생성한 프로그램의 실행이 종료되더라도 데이터가 사라지지 않고 지속적으로 유지되는 특성을 말합니다.

영속성은 주로 데이터베이스와 관련하여 사용되며, 데이터를 영구적으로 보존하고 유지하기 위한 개념입니다. 프로그램이 데이터를 생성하고 처리한 후에도 데이터는 메모리나 디스크 등의 저장 매체에 저장되어 지속적으로 사용될 수 있습니다.

영속성은 객체 지향 프로그래밍에서도 중요한 개념입니다. 객체를 생성하고 메모리에 유지하기 위해 영속성을 사용합니다. 객체의 상태를 유지하고 관리하기 위해 영속성 컨텍스트라는 개념을 사용하여 객체를 데이터베이스와 연결하고 데이터의 상태 변화를 추적합니다.

JPA(Java Persistence API)는 자바에서 영속성을 관리하기 위한 API이며, 객체와 관계형 데이터베이스 간의 매핑 및 영속성 관리를 위한 기능을 제공합니다. JPA를 사용하면 객체를 데이터베이스에 저장하고 조회하는 작업을 간편하게 처리할 수 있습니다.

영속성을 활용하면 데이터를 지속적으로 유지하고 필요한 시점에 데이터를 검색하거나 변경할 수 있으므로 데이터의 안정성과 일관성을 보장할 수 있습니다.

<ResponseEntity>
ResponseEntity는 Spring 프레임워크에서 HTTP 응답을 나타내는 클래스입니다. 이 클래스는 HTTP 응답의 상태 코드, 헤더, 본문 등을 포함할 수 있습니다.
ResponseEntity는 제네릭 타입으로 선언되어 있으며, 반환할 응답 본문의 타입을 명시할 수 있습니다. 일반적으로 ResponseEntity<T> 형태로 사용되며, T는 응답 본문의 데이터 타입을 나타냅니다.

ResponseEntity는 다양한 상태 코드를 포함할 수 있으며, 일반적으로 HTTP 응답의 상태 코드와 함께 사용됩니다. 예를 들어, ResponseEntity.ok()는 HTTP 상태 코드 200(OK)를 나타내는 응답을 생성합니다.
ResponseEntity를 사용하여 컨트롤러에서 클라이언트에게 응답을 보낼 수 있습니다. 이를 통해 응

답의 상태 코드, 헤더, 본문을 자유롭게 설정하고 전달할 수 있습니다.

<@JsonIgnore>
@JsonIgnore는 Jackson 라이브러리에서 제공하는 어노테이션입니다. 이 어노테이션을 사용하면 특정 필드나 메서드를 JSON 직렬화/역직렬화 과정에서 무시할 수 있습니다.

@JsonIgnore를 필드 위에 선언하면 해당 필드는 JSON으로 변환될 때 무시되며, 역직렬화 과정에서는 해당 필드에 값을 설정하지 않습니다. 마찬가지로 @JsonIgnore를 메서드(게터/세터) 위에 선언하면 해당 메서드를 통해 접근하는 필드가 JSON 변환에서 무시됩니다.

이 어노테이션은 주로 불필요한 데이터의 노출을 방지하고 JSON 직렬화/역직렬화 과정에서 특정 필드를 제외하고 싶을 때 사용됩니다. 예를 들어, 보안 관련 정보나 순환 참조를 방지하기 위해 사용할 수 있습니다.

@JsonIgnore를 사용하면 해당 필드나 메서드는 JSON으로 변환되지 않기 때문에 주의가 필요합니다. 필요에 따라서는 특정 상황에서 필드를 제외하고 싶을 때 사용하면 유용하게 활용할 수 있습니다.

<@Modifying>
@Modifying은 Spring Data JPA에서 사용되는 어노테이션 중 하나입니다. 이 어노테이션은 데이터를 수정하는 작업에 사용됩니다.

@Modifying 어노테이션은 다음과 같은 역할을 합니다:

@Query 어노테이션과 함께 사용되어 데이터베이스 쿼리를 실행할 때 수정 작업을 수행함을 나타냅니다.
수정 작업에 해당하는 쿼리 메서드를 선언할 때 필요합니다.
기본적으로 @Modifying 어노테이션이 지정된 메서드는 트랜잭션 내에서 실행되어야 합니다.
@Modifying 어노테이션은 다음과 같은 메서드에서 사용될 수 있습니다:

void를 반환하는 메서드: 데이터베이스의 INSERT, UPDATE, DELETE 작업을 수행할 때 사용됩니다.
int 또는 long을 반환하는 메서드: 수정 작업의 영향을 받는 레코드 수를 반환합니다.
예를 들어, 다음과 같은 메서드는 @Modifying 어노테이션과 함께 사용됩니다:

```
@Modifying
@Query("UPDATE User SET name = :name WHERE id = :id")
void updateUserName(@Param("id") Long id, @Param("name") String name);
```

위의 코드는 User 엔티티의 이름을 업데이트하는 쿼리를 실행하는 메서드입니다. @Modifying 어노테이션이 지정되어 있으므로 수정 작업을 수행하는 쿼리임을 나타냅니다.

<@ManyToOne 어노테이션의 fetch 속성>
@ManyToOne(fetch = FetchType.EAGER)는 JPA에서 사용되는 어노테이션입니다. 이 어노테이션은 연관 관계의 한쪽에 해당하는 엔티티가 다른 엔티티와의 관계를 설정하는데 사용됩니다.

@ManyToOne은 다대일(Many-to-One) 관계를 표현하는데 사용되며, 한 엔티티가 다른 엔티티에 대해 다중으로 참조될 수 있는 관계입니다. 일반적으로 외래 키(Foreign Key)를 사용하여 관계를 맺습니다. @ManyToOne 어노테이션을 사용하면 해당 필드가 다른 엔티티와의 관계를 나타내는 것을 알 수 있습니다.

fetch 속성은 데이터를 로딩할 때의 전략을 설정하는 데 사용됩니다. FetchType.EAGER는 즉시(Eager) 로딩을 의미합니다. 이 경우, 연관된 엔티티가 항상 즉시 로딩되어야 하며, 해당 엔티티를 조회하는 쿼리가 실행될 때 즉시 연관된 엔티티도 함께 로딩됩니다.

즉시 로딩은 연관된 엔티티가 항상 필요한 경우에 유용합니다. 그러나 성능상의 이슈가 발생할 수 있으며, 필요하지 않은 경우에는 지연 로딩(Lazy Loading) 전략을 고려해야 합니다.

예를 들어, 다음과 같이 @ManyToOne 어노테이션과 fetch 속성을 사용할 수 있습니다:

```
@ManyToOne(fetch = FetchType.EAGER)
private User user;
```
위의 예시에서는 User 엔티티와의 다대일 관계를 표현하며, fetch 속성이 FetchType.EAGER로 설정되어 해당 필드는 항상 즉시 로딩됩니다.

@ManyToOne(fetch = FetchType.LAZY)는 JPA에서 사용되는 어노테이션입니다. 이 어노테이션은 연관 관계의 한쪽에 해당하는 엔티티가 다른 엔티티와의 관계를 설정하는데 사용됩니다.

@ManyToOne은 다대일(Many-to-One) 관계를 표현하는데 사용되며, 한 엔티티가 다른 엔티티에 대해 다중으로 참조될 수 있는 관계입니다. 일반적으로 외래 키(Foreign Key)를 사용하여 관계를 맺습니다. @ManyToOne 어노테이션을 사용하면 해당 필드가 다른 엔티티와의 관계를 나타내는 것을 알 수 있습니다.

fetch 속성은 데이터를 로딩할 때의 전략을 설정하는 데 사용됩니다. FetchType.LAZY는 지연(Lazy) 로딩을 의미합니다. 이 경우, 연관된 엔티티는 실제로 필요한 시점에 로딩되며, 연관된 엔티티에 접근할 때 쿼리가 실행되어 로딩됩니다.

지연 로딩은 연관된 엔티티가 필요한 경우에만 로딩되므로 성능상의 이점을 가져올 수 있습니다. 그러나 실제로 연관된 엔티티에 접근할 때 쿼리가 실행되므로, 지연 로딩을 사용하는 경우 N+1 쿼리 문제에 유의해야 합니다.

예를 들어, 다음과 같이 @ManyToOne 어노테이션과 fetch 속성을 사용할 수 있습니다:

```
@ManyToOne(fetch = FetchType.LAZY)
private User user;
```
위의 예시에서는 User 엔티티와의 다대일 관계를 표현하며, fetch 속성이 FetchType.LAZY로 설정되어 해당 필드는 지연 로딩됩니다. 즉, user 필드에 접근할 때 실제로 로딩되고 쿼리가 실행됩니다.

<JPQL>
PQL(Java Persistence Query Language)은 JPA(Java Persistence API)에서 사용되는 객체 지향 쿼리 언어입니다. JPQL은 엔티티 객체를 대상으로 쿼리를 작성하고 실행하기 위해 사용됩니다.

JPQL은 SQL과는 달리 엔티티 객체를 대상으로 쿼리를 작성하기 때문에 객체 지향적인 접근 방식을 지원합니다. JPQL은 엔티티와 관련된 속성, 연관 관계, 상속 구조 등을 고려하여 쿼리를 작성할 수 있으며, SQL과 유사한 문법을 사용합니다.

JPQL은 데이터베이스의 구체적인 구현과는 독립적이며, JPA가 지원하는 모든 데이터베이스에서 사용할 수 있습니다. JPQL은 엔티티 객체를 대상으로 쿼리를 작성하고 실행하기 때문에 객체 그래프를 통해 데이터를 조회하거나 조작할 수 있습니다.

JPQL은 EntityManager를 통해 실행되며, EntityManager는 JPA의 핵심 인터페이스입니다. JPQL을 사용하여 엔티티 객체를 조회하거나 조작할 수 있으며, 결과는 엔티티 객체 또는 엔티티 컬렉션으로 반환됩니다.

JPQL은 SQL과 유사한 문법을 사용하지만, 테이블이 아닌 엔티티와 속성을 기준으로 쿼리를 작성하므로 객체 지향적인 개발에 적합한 쿼리 언어입니다.
나머지는 소스 코드 참조

파일 업로드 하기, Thymeleaf 개요와 Thymeleaf 활용하기

<파일 업로드>

```
<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
```

<@Transient>
JPA에서 필드를 영속성 컨텍스트에 저장하지 않도록 지정하는 어노테이션입니다.

<Thymeleaf>
Thymeleaf는 Java 및 Spring 기반 웹 애플리케이션에서 사용할 수 있는 서버 사이드 Java 템플릿 엔진입니다. HTML, XML, JavaScript, CSS 등의 마크업 언어를 지원하며, 동적 데이터와 서버 사이드 로직을 템플릿에 쉽게 통합할 수 있습니다.

Thymeleaf는 일반적인 HTML 파일로도 읽을 수 있으며, 템플릿 속성에 추가된 Thymeleaf 특수 문법을 사용하여 동적으로 템플릿을 렌더링할 수 있습니다. 이 특수 문법은 HTML 요소에 태그, 속성 및 표현식을 추가하는 방식으로 작동합니다. Thymeleaf는 템플릿과 데이터를 결합하여 완전한 HTML 문서를 생성하는 데 사용됩니다.

Spring Framework와 통합되어 Spring MVC와 함께 사용할 때 Thymeleaf는 컨트롤러에서 전달되는 데이터를 템플릿과 결합하여 동적으로 웹 페이지를 생성하는 데 사용됩니다. Thymeleaf는 Spring MVC의 View Resolver로 구성할 수 있으며, 컨트롤러의 메소드에서 반환된 데이터와 템플릿을 결합하여 클라이언트에게 전송될 최종 HTML을 생성합니다.

Thymeleaf는 사용하기 쉽고 직관적인 문법을 제공하며, Spring 프로젝트에서 널리 사용되는 템플릿 엔진 중 하나입니다.

<Thymeleaf 설정>

config 파일
server.port=7755

#database
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url
=jdbc:mysql://localhost:3306/springdb?useSSL=false&serverTimezone=Asia/Seoul&characterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password=root

spring.devtools.livereload.enabled=true

pom.xml 추가할 것 없음.
Thymeleaf Dependency에 추가 되어 있는거 확인(Starter 만들 때 체크하면 있음)
mvn Repository - thymeleaf
<!-- https://mvnrepository.com/artifact/nz.net.ultraq.thymeleaf/thymeleaf-layout-dialect -->
<dependency>
    <groupId>nz.net.ultraq.thymeleaf</groupId>
    <artifactId>thymeleaf-layout-dialect</artifactId>
</dependency>
복사 붙여넣어주기 pom.xml

Controller, DTO, Service 패키지 생성
홈컨트롤러 생성
templates에 home.html
<home.html>
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
layout:decorate="~{layouts/layout1}"></html>

<템플릿에 폴더 생성-layouts-layout1.html 생성>
<layout1.html>
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
<meta charset="UTF-8">
<title>NEW HTML</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/css/bootstrap.min.css">
<script
        src="https://cdn.jsdelivr.net/npm/jquery@3.6.4/dist/jquery.slim.min.js"></script>
<script
        src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/popper.min.js"></script>
<script

```
src="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/js/bootstrap.bundle.min.js"></script>
<script src="https://code.jquery.com/jquery-3.7.0.min.js"></script>
</head>

<body>
	<div th:replace = "fragments/header::header"></div>
	<div layout:fragment="content" class="content"></div>
	<!--<div th:replace = "fragments/footer::footer"></div>-->
</body>
</html>
```

<fragments 폴더 생성 - header.html 생성(footer 생성은 해도 되지만 생략함)>

<header.html>

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<div th:fragment = "header">
	<nav class="navbar navbar-expand-sm bg-warning navbar-dark mb-3">

		<div class="container mt-3">
			<!-- Brand/logo -->
			<a class="navbar-brand" href="/">HOME(Board)</a>

			<!-- Links -->
			<ul class="navbar-nav mr-auto">
				<li class="nav-item"><a class="nav-link"
href="/board/insert">BoardInsert</a></li>
				<li class="nav-item"><a class="nav-link"
href="/board/list">BoardList</a></li>
			</ul>

			<ul class="navbar-nav">

				<li class="nav-item"><a class="nav-link"
href="/login">로그인</a></li>
				<li class="nav-item"><a class="nav-link"
href="/join">회원가입</a></li>


				<li class="nav-item">
				<a class="nav-link" href="/logout">
				로그아웃</a></li>
			</ul>

		</div>
	</nav>
</div>
```

나머지도 작성해주면 됨 - 소스 코드 참조

<회원 리스트까지 작성>
소스 코드 참조

<BuilderTest - Junit 활용>

```
package com.builder.test01;

public class User {
	private String name;
	private int age;

	public static UserBuilder builder() {
		return new UserBuilder();
	}

	// getter, setter
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

}
```

```
package com.builder.test01;

public class UserBuilder {
	private String name;
	private int age;

	public User build() {
		User user = new User();
		user.setAge(this.age);
		user.setName(this.name);
		return user;
	}

	public UserBuilder name(String name) {
```

```
                this.name = name;
                return this;
        }

        public UserBuilder age(int age) {
                this.age = age;
                return this;
        }
}

package com.builder.test01;

import org.junit.Test;

public class UserBuilderTest {
        @Test
        public void userbuilderTest() {
        User user =  User.builder()
                .name("홍길동")
                .age(20)
                .build();
        System.out.println(user);

        User user1 = User.builder()
                        .age(33)
                        .name("이순신")
                        .build();
        System.out.println(user1);


        }
}


package com.builder.test02;

import lombok.Builder;
import lombok.Getter;
import lombok.Setter;

@Getter @Setter
@Builder
public class User {
        private String name;
        private int age;
}


package com.builder.test02;
```

```java
import org.junit.Test;

public class UserBuilderTest02 {
	@Test
	public void builderTest02() {
		User user = User.builder()
				.name("홍길동")
				.age(22)
				.build();
		System.out.println(user);

		User user1 = User.builder()
				.age(44)
				.name("이자바")
				.build();
		System.out.println(user1);
	}



}
```

Junit, lombok : Module Path에 추가 해 줍니다.

Spring Boot Project Import
<이미 구현되어 있는 Spring Boot를 Import 하기>
마우스 우클릭 - Import - Projects from Folder and Archive 선택 - Next - shop-master.zip 압축 그대로 가져오기

가져온 후 생성됨.
생성 후 생성된 프로젝트에서 마우스 오른쪽 클릭 - Maven - Update Project 선택해서 Maven Update 합니다.
리소스 application.properties 내용 수정, application.properties 마우스 우클릭 - 글자 깨지니 UTF-8로 변경, DB부분(password 부분)도 수정

```
#애플리케이션 포트 설정
server.port = 80

#MySQL 연결 설정
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/shop?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root <!-- 1234로 지정되어 있는 것 변경 -->

#실행되는 쿼리 콘솔 출력
spring.jpa.properties.hibernate.show_sql=true

#콘솔창에 출력되는 쿼리를 가독성이 좋게 포맷팅
```

```
spring.jpa.properties.hibernate.format_sql=true

#쿼리에 물음표로 출력되는 바인드 파라미터 출력
logging.level.org.hibernate.type.descriptor.sql=trace

spring.jpa.hibernate.ddl-auto=update
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

#Live Reload 기능 활성화
spring.devtools.livereload.enabled=true

#Thymeleaf cache 사용 중지
spring.thymeleaf.cache = false

#파일 한 개당 최대 사이즈
spring.servlet.multipart.maxFileSize=20MB
#요청당 최대 파일 크기
spring.servlet.multipart.maxRequestSize=100MB
#상품 이미지 업로드 경로
itemImgLocation=C:/shop/item   <!-- 경로 생성해줘야 함-->
#리소스 업로드 경로
uploadPath=file:///C:/shop/ <!-- 경로 생성해줘야 함-->

#기본 batch size 설정
spring.jpa.properties.hibernate.default_batch_fetch_size=1000
```

경로 생성해줘야 함.
C:/shop/item
C:/shop/

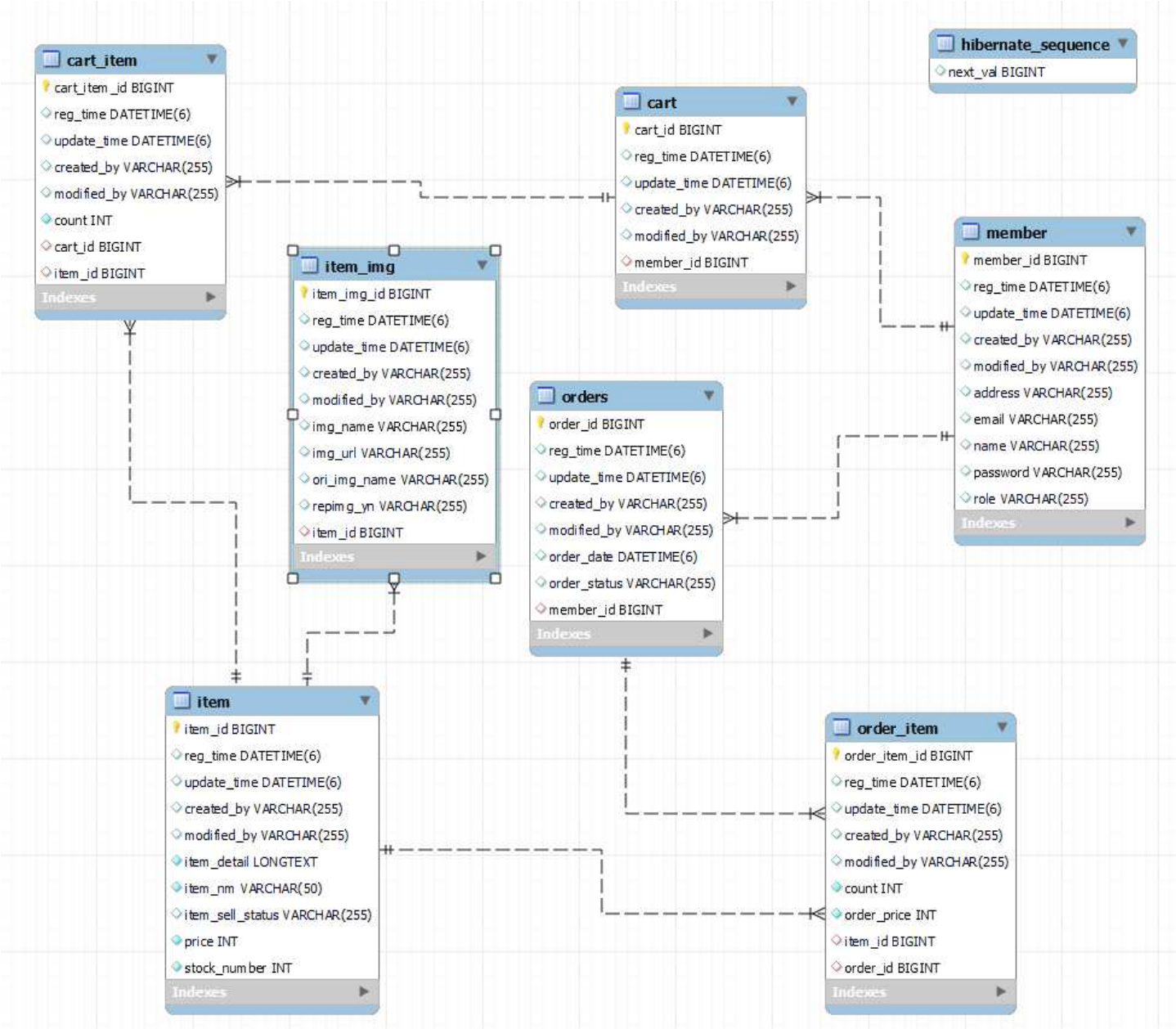
DB(MySQL)에서 스키마 생성(Create Schema) 하기 - 이름 : shop

나머지는 파일 참조(기존에 수업해오면서 해왔던 환경과 유사함.)

<QueryDSL>
QueryDSL은 자바 기반의 동적 쿼리를 생성하기 위한 라이브러리로, SQL 쿼리를 직접 작성하는 대신 자바 코드로 쿼리를 작성할 수 있도록 도와줍니다.

QueryDSL을 사용하면 컴파일 시점에서 쿼리 문법의 오류를 찾을 수 있고, IDE의 자동 완성 기능을 활용하여 쿼리 작성이 용이해집니다. 또한, 동적 쿼리를 쉽게 구성할 수 있어 유지보수성과 가독성을 향상시킬 수 있습니다.

<MySQL Reverse Engineering을 활용한 E-R 모델링 관계>


cart_item
cart_item_id BIGINT
reg_time DATETIME(6)
update_time DATETIME(6)
created_by VARCHAR(255)
modified_by VARCHAR(255)
count INT
cart_id BIGINT
item_id BIGINT
Indexes
hibernate_sequence
next_val BIGINT
cart
cart_id BIGINT
reg_time DATETIME(6)
update_time DATETIME(6)
created_by VARCHAR(255)
modified_by VARCHAR(255)
member_id BIGINT
Indexes
member
member_id BIGINT
reg_time DATETIME(6)
update_time DATETIME(6)
created_by VARCHAR(255)
modified_by VARCHAR(255)
address VARCHAR(255)
email VARCHAR(255)
name VARCHAR(255)
password VARCHAR(255)
role VARCHAR(255)
Indexes
item_img
item_img_id BIGINT
reg_time DATETIME(6)
update_time DATETIME(6)
created_by VARCHAR(255)
modified_by VARCHAR(255)
img_name VARCHAR(255)
img_url VARCHAR(255)
ori_img_name VARCHAR(255)
repimg_yn VARCHAR(255)
item_id BIGINT
Indexes
orders
order_id BIGINT
reg_time DATETIME(6)
update_time DATETIME(6)
created_by VARCHAR(255)
modified_by VARCHAR(255)
order_date DATETIME(6)
order_status VARCHAR(255)
member_id BIGINT
Indexes
item
item_id BIGINT
reg_time DATETIME(6)
update_time DATETIME(6)
created_by VARCHAR(255)
modified_by VARCHAR(255)
item_detail LONGTEXT
item_nm VARCHAR(50)
item_sell_status VARCHAR(255)
price INT
stock_number INT
Indexes
order_item
order_item_id BIGINT
reg_time DATETIME(6)
update_time DATETIME(6)
created_by VARCHAR(255)
modified_by VARCHAR(255)
count INT
order_price INT
item_id BIGINT
order_id BIGINT
Indexes