

## 6. 스프링 DB 접근 기술

#1.인강/1.스프링 입문/강의#

- /H2 데이터베이스 설치
- /순수 Jdbc
- /스프링 통합 테스트
- /스프링 JdbcTemplate
- /JPA
- /스프링 데이터 JPA

### H2 데이터베이스 설치

개발이나 테스트 용도로 가볍고 편리한 DB, 웹 화면 제공

- <https://www.h2database.com>
- 다운로드 및 설치
- h2 데이터베이스 버전은 스프링 부트 버전에 맞춘다.
- 권한 주기: `chmod 755 h2.sh` (윈도우 사용자는 x)
- 실행: `./h2.sh` (윈도우 사용자는 `h2.bat`)
- 데이터베이스 파일 생성 방법
  - `jdbc:h2:~/test` (최초 한번)
  - `~/test.mv.db` 파일 생성 확인
  - 이후부터는 `jdbc:h2:tcp://localhost/~/test` 이렇게 접속

**h2 데이터베이스 버전은 스프링 부트 버전에 맞춘다.**

- 스프링 부트 2.x를 사용하면 **1.4.200 버전**을 다운로드 받으면 된다.
- 스프링 부트 3.x를 사용하면 **2.1.214 버전 이상** 사용해야 한다.
- 다음 링크에 가면 다양한 H2 다운로드 버전을 확인할 수 있다.
- <https://www.h2database.com/html/download-archive.html>

### 테이블 생성하기

테이블 관리를 위해 프로젝트 루트에 `sql/ddl.sql` 파일을 생성

```
drop table if exists member CASCADE;  
create table member  
(
```

```
id bigint generated by default as identity,  
name varchar(255),  
primary key (id)  
);
```

H2 데이터베이스에 접근해서 `member` 테이블 생성


## H2 데이터베이스가 정상 생성되지 않을 때

다음과 같은 오류 메시지가 나오며 H2 데이터베이스가 정상 생성되지 않는 경우가 있다.


Database "/Users/kimyoungha/test" not found, either pre-create it or allow remote database creation (not recommended in secure environments) [90149-200] 90149/90149 (도움말)

해결방안은 다음과 같다.

1. H2 데이터베이스를 종료하고, 다시 시작한다.
2. 웹 브라우저가 자동 실행되면 주소창에 다음과 같이 되어있다.(100.1.2.3이 아니라 임의의 숫자가 나온다.)

 100.1.2.3:8082/login.jsp?jsessionid=d1e9d744ce01f475cb1c0bce286be109

3. 다음과 같이 앞 부분만 `100.1.2.3` → `localhost` 로 변경하고 Enter를 입력한다. 나머지 부분은 절대 변경하면 안된다. (특히 뒤에 세션 부분이 변경되면 안된다.)

 localhost:8082/login.jsp?jsessionid=d1e9d744ce01f475cb1c0bce286be109

4. 데이터베이스 파일을 생성하면(`jdbc:h2:~/test`), 데이터베이스가 정상 생성된다.

## 순수 Jdbc

### 환경 설정

`build.gradle` 파일에 `jdbc`, `h2` 데이터베이스 관련 라이브러리 추가

```
implementation 'org.springframework.boot:spring-boot-starter-jdbc'  
runtimeOnly 'com.h2database:h2'
```

## 스프링 부트 데이터베이스 연결 설정 추가

resources/application.properties

```
spring.datasource.url=jdbc:h2:tcp://localhost/~test
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
```

**주의!** 스프링부트 2.4부터는 `spring.datasource.username=sa` 를 꼭 추가해주어야 한다. 그렇지 않으면 `Wrong user name or password` 오류가 발생한다. 참고로 다음과 같이 마지막에 공백이 들어가면 같은 오류가 발생한다. `spring.datasource.username=sa` ← 공백 주의, 공백은 모두 제거해야 한다.

참고: 인텔리J 커뮤니티(무료) 버전의 경우 `application.properties` 파일의 왼쪽이 다음 그림과 같이 회색으로 나온다. 엔터프라이즈(유료) 버전에서 제공하는 스프링의 소스 코드를 연결해주는 편의 기능이 빠진 것인데, 실제 동작하는데는 아무런 문제가 없다.

```
spring.datasource.url=jdbc:h2:tcp://localhost/~test
spring.datasource.driver-class-name=org.h2.Driver
```

## Jdbc 리포지토리 구현

주의! 이렇게 JDBC API로 직접 코딩하는 것은 20년 전 이야기이다. 따라서 고대 개발자들이 이렇게 고생하고 살았구나 생각하고, 정신건강을 위해 참고만 하고 넘어가자.

### Jdbc 회원 리포지토리

```
package hello.hellospring.repository;

import hello.hellospring.domain.Member;
import org.springframework.jdbc.datasource.DataSourceUtils;

import javax.sql.DataSource;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
```

```

public class JdbcMemberRepository implements MemberRepository {

    private final DataSource dataSource;

    public JdbcMemberRepository(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public Member save(Member member) {
        String sql = "insert into member(name) values(?)";

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            conn = getConnection();
            pstmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);

            pstmt.setString(1, member.getName());

            pstmt.executeUpdate();
            rs = pstmt.getGeneratedKeys();

            if (rs.next()) {
                member.setId(rs.getLong(1));
            } else {
                throw new SQLException("id 조회 실패");
            }
            return member;
        } catch (Exception e) {
            throw new IllegalStateException(e);
        } finally {
            close(conn, pstmt, rs);
        }
    }

    @Override
    public Optional<Member> findById(Long id) {
        String sql = "select * from member where id = ?";

        Connection conn = null;
    }

```

```

PreparedStatement pstmt = null;
ResultSet rs = null;

try {
    conn = getConnection();
    pstmt = conn.prepareStatement(sql);
    pstmt.setLong(1, id);

    rs = pstmt.executeQuery();

    if(rs.next()) {
        Member member = new Member();
        member.setId(rs.getLong("id"));
        member.setName(rs.getString("name"));
        return Optional.of(member);
    } else {
        return Optional.empty();
    }

} catch (Exception e) {
    throw new IllegalStateException(e);
} finally {
    close(conn, pstmt, rs);
}

}

```

@Override

```

public List<Member> findAll() {
    String sql = "select * from member";

    Connection conn = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;

    try {
        conn = getConnection();
        pstmt = conn.prepareStatement(sql);

        rs = pstmt.executeQuery();

        List<Member> members = new ArrayList<>();
        while(rs.next()) {

```

```

        Member member = new Member();
        member.setId(rs.getLong("id"));
        member.setName(rs.getString("name"));
        members.add(member);
    }

    return members;
} catch (Exception e) {
    throw new IllegalStateException(e);
} finally {
    close(conn, pstmt, rs);
}
}

@Override
public Optional<Member> findByName(String name) {
    String sql = "select * from member where name = ?";

    Connection conn = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;

    try {
        conn = getConnection();
        pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, name);

        rs = pstmt.executeQuery();

        if(rs.next()) {
            Member member = new Member();
            member.setId(rs.getLong("id"));
            member.setName(rs.getString("name"));
            return Optional.of(member);
        }

        return Optional.empty();
    } catch (Exception e) {
        throw new IllegalStateException(e);
    } finally {
        close(conn, pstmt, rs);
    }
}
}

```

```

private Connection getConnection() {
    return DataSourceUtils.getConnection(dataSource);
}

private void close(Connection conn, PreparedStatement pstmt, ResultSet rs) {
    try {
        if (rs != null) {
            rs.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    try {
        if (pstmt != null) {
            pstmt.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    try {
        if (conn != null) {
            close(conn);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

private void close(Connection conn) throws SQLException {
    DataSourceUtils.releaseConnection(conn, dataSource);
}
}

```

## 스프링 설정 변경

```

package hello.hellospring;

import hello.hellospring.repository.JdbcMemberRepository;
import hello.hellospring.repository.JdbcTemplateMemberRepository;
import hello.hellospring.repository.MemberRepository;
import hello.hellospring.repository.MemoryMemberRepository;

```

```

import hello.hellospring.service.MemberService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration
public class SpringConfig {

    private final DataSource dataSource;

    public SpringConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public MemberService memberService() {
        return new MemberService(memberRepository());
    }

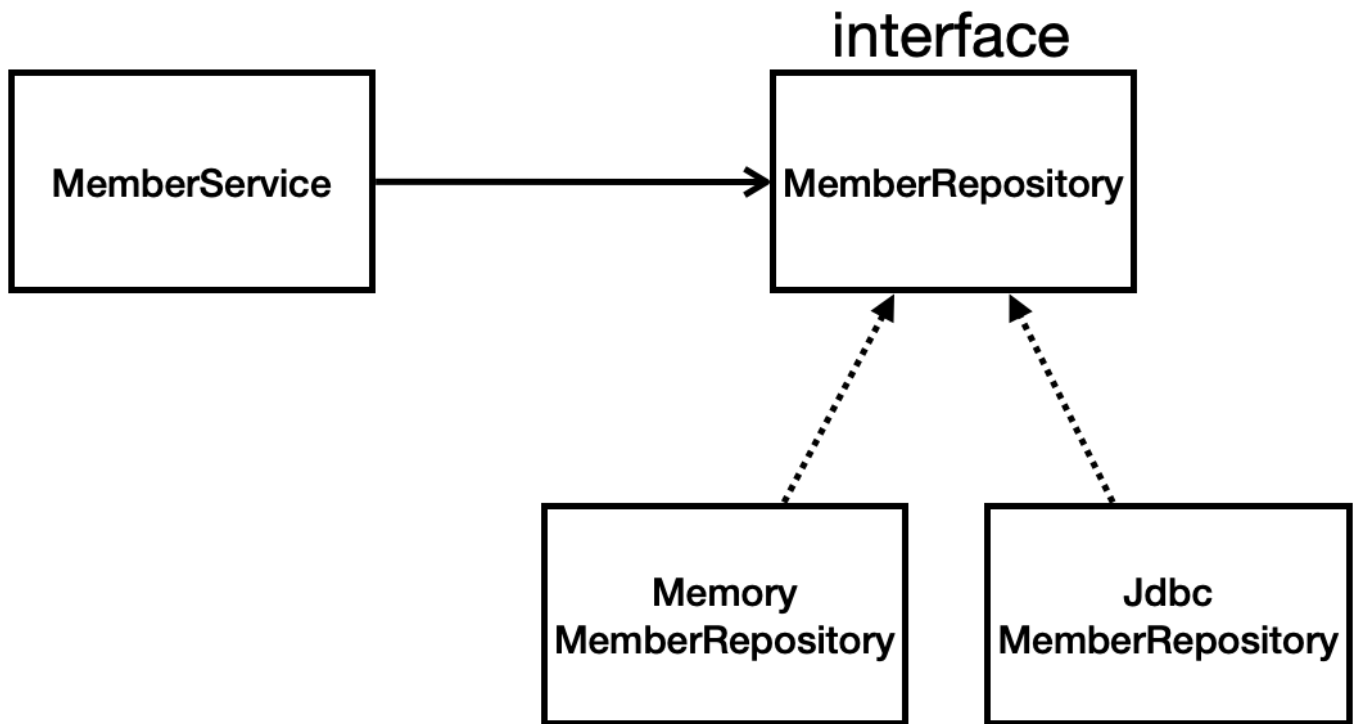
    @Bean
    public MemberRepository memberRepository() {
        // return new MemoryMemberRepository();
        return new JdbcMemberRepository(dataSource);
    }
}

```

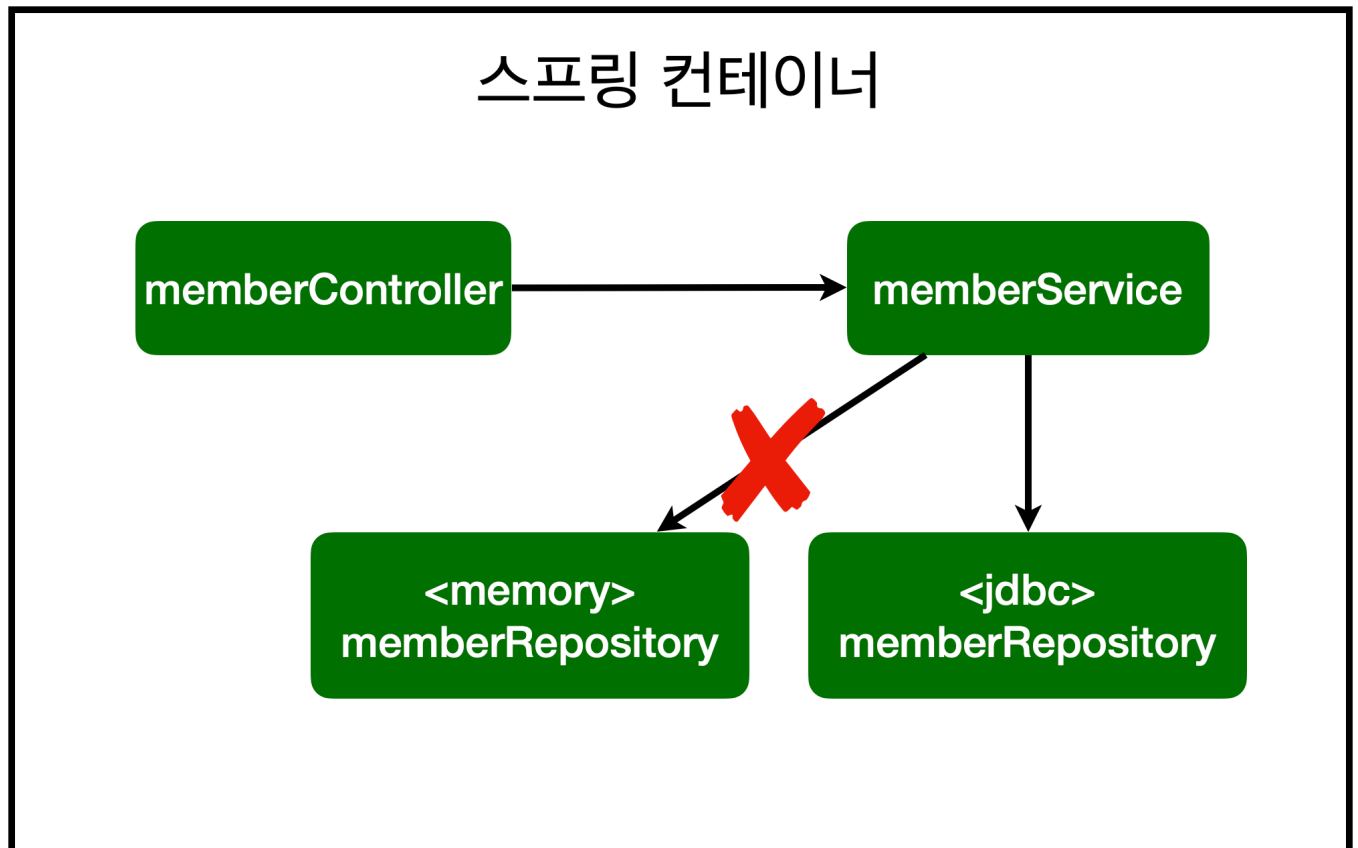
- DataSource는 데이터베이스 커넥션을 획득할 때 사용하는 객체다. 스프링 부트는 데이터베이스 커넥션 정보를 바탕으로 DataSource를 생성하고 스프링 빈으로 만들어둔다. 그래서 DI를 받을 수 있다.

구현 클래스 추가 이미지





스프링 설정 이미지



- 개방-폐쇄 원칙(OCP, Open-Closed Principle)
  - 확장에는 열려있고, 수정, 변경에는 닫혀있다.
- 스프링의 DI (Dependencies Injection)을 사용하면 기존 코드를 전혀 손대지 않고, 설정만으로 구현 클래스를 변경할 수 있다.

- 회원을 등록하고 DB에 결과가 잘 입력되는지 확인하자.
- 데이터를 DB에 저장하므로 스프링 서버를 다시 실행해도 데이터가 안전하게 저장된다.

## 스프링 통합 테스트

스프링 컨테이너와 DB까지 연결한 통합 테스트를 진행해보자.

### 회원 서비스 스프링 통합 테스트

```
package hello.hellospring.service;

import hello.hellospring.domain.Member;
import hello.hellospring.repository.MemberRepository;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;

import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

@SpringBootTest
@Transactional
class MemberServiceIntegrationTest {

    @Autowired MemberService memberService;
    @Autowired MemberRepository memberRepository;

    @Test
    public void 회원가입() throws Exception {

        //Given
        Member member = new Member();
        member.setName("hello");

        //When
        Long saveId = memberService.join(member);
```

```

        //Then
        Member findMember = memberRepository.findById(saveId).get();
        assertEquals(member.getName(), findMember.getName());
    }

    @Test
    public void 중복_회원_예외() throws Exception {
        //Given
        Member member1 = new Member();
        member1.setName("spring");

        Member member2 = new Member();
        member2.setName("spring");

        //When
        memberService.join(member1);
        IllegalStateException e = assertThrows(IllegalStateException.class,
            () -> memberService.join(member2)); //예외가 발생해야 한다.

        assertEquals(e.getMessage(), "이미 존재하는 회원입니다.");
    }
}

```

- `@SpringBootTest`: 스프링 컨테이너와 테스트를 함께 실행한다.
- `@Transactional`: 테스트 케이스에 이 애노테이션이 있으면, 테스트 시작 전에 트랜잭션을 시작하고, 테스트 완료 후에 항상 롤백한다. 이렇게 하면 DB에 데이터가 남지 않으므로 다음 테스트에 영향을 주지 않는다.

## 스프링 JdbcTemplate

- 순수 Jdbc와 동일한 환경설정을 하면 된다.
- 스프링 JdbcTemplate과 MyBatis 같은 라이브러리는 JDBC API에서 본 반복 코드를 대부분 제거해준다. 하지만 SQL은 직접 작성해야 한다.

### 스프링 JdbcTemplate 회원 리포지토리

```

package hello.hellospring.repository;

```

```

import hello.hellospring.domain.Member;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;

import javax.sql.DataSource;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;

public class JdbcTemplateMemberRepository implements MemberRepository {

    private final JdbcTemplate jdbcTemplate;

    public JdbcTemplateMemberRepository(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Override
    public Member save(Member member) {
        SimpleJdbcInsert jdbcInsert = new SimpleJdbcInsert(jdbcTemplate);
        jdbcInsert.withTableName("member").usingGeneratedKeyColumns("id");

        Map<String, Object> parameters = new HashMap<>();
        parameters.put("name", member.getName());

        Number key = jdbcInsert.executeAndReturnKey(new
        MapSqlParameterSource(parameters));
        member.setId(key.longValue());
        return member;
    }

    @Override
    public Optional<Member> findById(Long id) {
        List<Member> result = jdbcTemplate.query("select * from member where id
        = ?", memberRowMapper(), id);
        return result.stream().findAny();
    }
}

```

```

@Override
public List<Member> findAll() {
    return jdbcTemplate.query("select * from member", memberRowMapper());
}

@Override
public Optional<Member> findByName(String name) {
    List<Member> result = jdbcTemplate.query("select * from member where
name = ?", memberRowMapper(), name);
    return result.stream().findAny();
}

private RowMapper<Member> memberRowMapper() {
    return (rs, rowNum) -> {
        Member member = new Member();
        member.setId(rs.getLong("id"));
        member.setName(rs.getString("name"));
        return member;
    };
}
}

```

## JdbcTemplate을 사용하도록 스프링 설정 변경

```

package hello.hellospring;

import hello.hellospring.repository.JdbcMemberRepository;
import hello.hellospring.repository.JdbcTemplateMemberRepository;
import hello.hellospring.repository.MemberRepository;
import hello.hellospring.repository.MemoryMemberRepository;
import hello.hellospring.service.MemberService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration
public class SpringConfig {

    private final DataSource dataSource;

```

```

public SpringConfig(dataSource) {
    this.dataSource = dataSource;
}

@Bean
public MemberService memberService() {
    return new MemberService(memberRepository());
}

@Bean
public MemberRepository memberRepository() {
    // return new MemoryMemberRepository();
    // return new JdbcMemberRepository(dataSource);
    return new JdbcTemplateMemberRepository(dataSource);
}
}

```

## JPA

- JPA는 기존의 반복 코드는 물론이고, 기본적인 SQL도 JPA가 직접 만들어서 실행해준다.
- JPA를 사용하면, SQL과 데이터 중심의 설계에서 객체 중심의 설계로 패러다임을 전환을 할 수 있다.
- JPA를 사용하면 개발 생산성을 크게 높일 수 있다.

### build.gradle 파일에 JPA, h2 데이터베이스 관련 라이브러리 추가

```

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    // implementation 'org.springframework.boot:spring-boot-starter-jdbc'
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    runtimeOnly 'com.h2database:h2'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}

```

spring-boot-starter-data-jpa는 내부에 jdbc 관련 라이브러리를 포함한다. 따라서 jdbc는 제거해도 된다.

## 스프링 부트에 JPA 설정 추가

```
resources/application.properties

spring.datasource.url=jdbc:h2:tcp://localhost/~ /test
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none
```

**주의!** 스프링부트 2.4부터는 `spring.datasource.username=sa` 를 꼭 추가해주어야 한다. 그렇지 않으면 오류가 발생한다.

- `show-sql`: JPA가 생성하는 SQL을 출력한다.
- `ddl-auto`: JPA는 테이블을 자동으로 생성하는 기능을 제공하는데 `none` 를 사용하면 해당 기능을 끈다.
  - `create` 를 사용하면 엔티티 정보를 바탕으로 테이블도 직접 생성해준다. → 해보자.

## JPA 엔티티 매핑

```
package hello.hellospring.domain;

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

@Entity
public class Member {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
```

```

        return name;
    }

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

```

## JPA 회원 리포지토리

```

package hello.hellospring.repository;

import hello.hellospring.domain.Member;

import javax.persistence.EntityManager;
import java.util.List;
import java.util.Optional;

public class JpaMemberRepository implements MemberRepository {

    private final EntityManager em;

    public JpaMemberRepository(EntityManager em) {
        this.em = em;
    }

    public Member save(Member member) {
        em.persist(member);
        return member;
    }

    public Optional<Member> findById(Long id) {
        Member member = em.find(Member.class, id);
        return Optional.ofNullable(member);
    }

    public List<Member> findAll() {
        return em.createQuery("select m from Member m", Member.class)
            .getResultList();
    }
}

```



```

    public Optional<Member> findByName(String name) {
        List<Member> result = em.createQuery("select m from Member m where
m.name = :name", Member.class)
            .setParameter("name", name)
            .getResultList();
        return result.stream().findAny();
    }
}

```

## 서비스 계층에 트랜잭션 추가

```

import org.springframework.transaction.annotation.Transactional

@Transactional
public class MemberService {}

```

- `org.springframework.transaction.annotation.Transactional` 를 사용하자.
- 스프링은 해당 클래스의 메서드를 실행할 때 트랜잭션을 시작하고, 메서드가 정상 종료되면 트랜잭션을 커밋한다. 만약 런타임 예외가 발생하면 롤백한다.
- **JPA를 통한 모든 데이터 변경은 트랜잭션 안에서 실행해야 한다.**

## JPA를 사용하도록 스프링 설정 변경

```

package hello.hellospring;

import hello.hellospring.repository.*;
import hello.hellospring.service.MemberService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.persistence.EntityManager;
import javax.sql.DataSource;

@Configuration
public class SpringConfig {

    private final DataSource dataSource;
    private final EntityManager em;

    public SpringConfig(DataSource dataSource, EntityManager em) {
        this.dataSource = dataSource;
    }
}

```

```

        this.em = em;
    }

    @Bean
    public MemberService memberService() {
        return new MemberService(memberRepository());
    }

    @Bean
    public MemberRepository memberRepository() {
        // return new MemoryMemberRepository();
        // return new JdbcMemberRepository(dataSource);
        // return new JdbcTemplateMemberRepository(dataSource);
        return new JpaMemberRepository(em);
    }
}

```

참고: JPA도 스프링 만큼 성숙한 기술이고, 학습해야 할 분량도 방대하다. 다음 강의와 책을 참고하자.

- 인프런 강의 링크: [인프런 - 자바 ORM 표준 JPA 프로그래밍 - 기본편](#)
- JPA 책 링크: [자바 ORM 표준 JPA 프로그래밍 - YES24](#)

## 스프링 데이터 JPA

스프링 부트와 JPA만 사용해도 개발 생산성이 정말 많이 증가하고, 개발해야 할 코드도 확연히 줄어듭니다. 여기에 스프링 데이터 JPA를 사용하면, 기존의 한계를 넘어 마치 마법처럼, 리포지토리에 구현 클래스 없이 인터페이스만으로 개발을 완료할 수 있습니다. 그리고 반복 개발해온 기본 CRUD 기능도 스프링 데이터 JPA가 모두 제공합니다.

스프링 부트와 JPA라는 기반 위에, 스프링 데이터 JPA라는 환상적인 프레임워크를 더하면 개발이 정말 즐거워집니다. 지금까지 조금이라도 단순하고 반복이라 생각했던 개발 코드들이 확연하게 줄어듭니다. 따라서 개발자는 핵심 비즈니스 로직을 개발하는데, 집중할 수 있습니다.

실무에서 관계형 데이터베이스를 사용한다면 스프링 데이터 JPA는 이제 선택이 아니라 필수입니다.

주의: 스프링 데이터 JPA는 JPA를 편리하게 사용하도록 도와주는 기술입니다. 따라서 JPA를 먼저 학습한 후에 스프링 데이터 JPA를 학습해야 합니다.

- 앞의 JPA 설정을 그대로 사용한다.

## 스프링 데이터 JPA 회원 리포지토리

```
package hello.hellospring.repository;

import hello.hellospring.domain.Member;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface SpringDataJpaMemberRepository extends JpaRepository<Member, Long>, MemberRepository {

    Optional<Member> findByName(String name);
}
```

## 스프링 데이터 JPA 회원 리포지토리를 사용하도록 스프링 설정 변경

```
package hello.hellospring;

import hello.hellospring.repository.*;
import hello.hellospring.service.MemberService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringConfig {

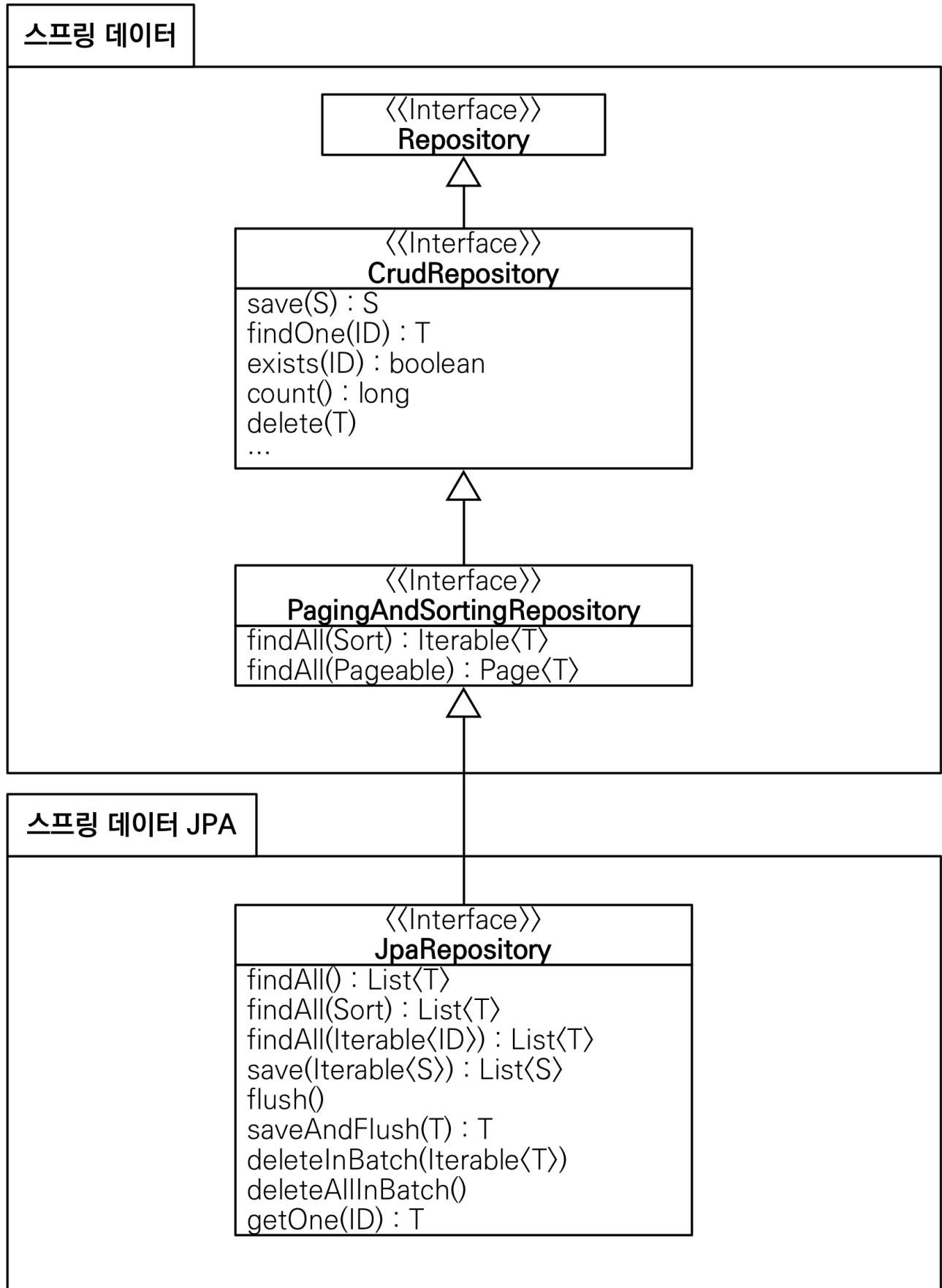
    private final MemberRepository memberRepository;

    public SpringConfig(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }

    @Bean
    public MemberService memberService() {
        return new MemberService(memberRepository);
    }
}
```

- 스프링 데이터 JPA가 `SpringDataJpaMemberRepository` 를 스프링 빈으로 자동 등록해준다.

## 스프링 데이터 JPA 제공 클래스



## 스프링 데이터 JPA 제공 기능

- 인터페이스를 통한 기본적인 CRUD
- `findByName()`, `findByEmail()` 처럼 메서드 이름만으로 조회 기능 제공
- 페이징 기능 자동 제공

참고: 실무에서는 JPA와 스프링 데이터 JPA를 기본으로 사용하고, 복잡한 동적 쿼리는 Querydsl이라는 라이브러리를 사용하면 된다. Querydsl을 사용하면 쿼리도 자바 코드로 안전하게 작성할 수 있고, 동적 쿼리도 편리하게 작성할 수 있다. 이 조합으로 해결하기 어려운 쿼리는 JPA가 제공하는 네이티브 쿼리를 사용하거나, 앞서 학습한 스프링 JdbcTemplate를 사용하면 된다.

자세한 내용은 다음 강의를 참고하자: [인프런 - 실전! 스프링 데이터 JPA](#)