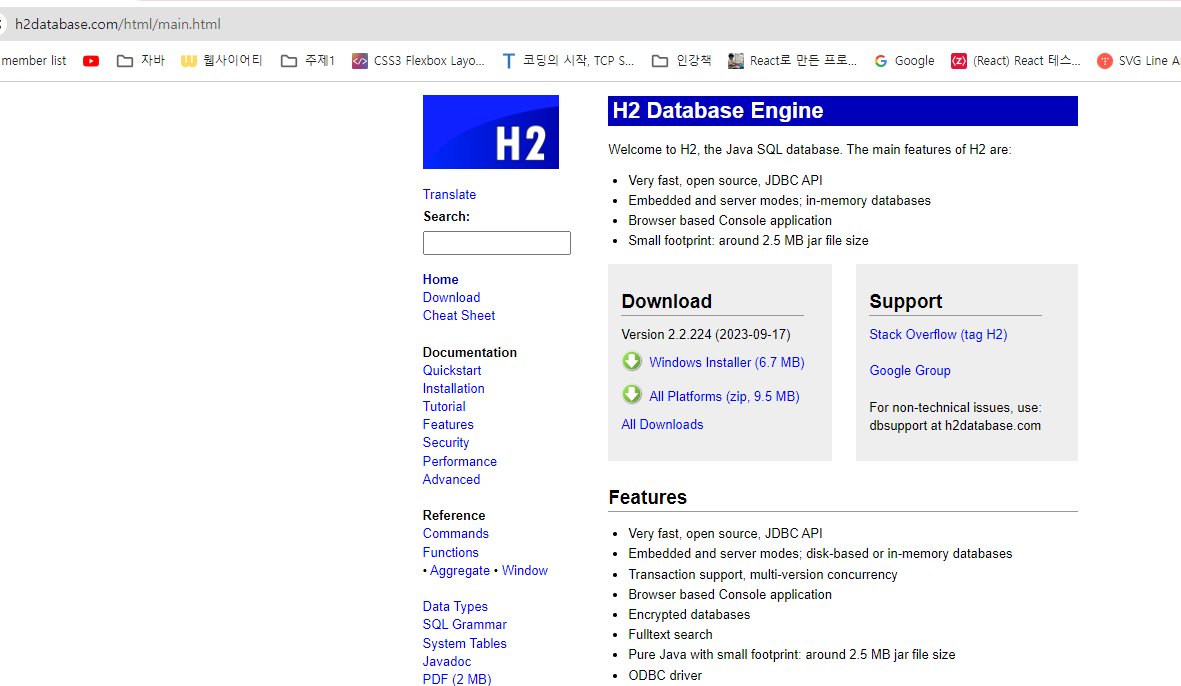
**스프링 DB 접근 기술**

**📒 H2 데이터베이스 설치**

**🖋️ H2 데이터베이스 설치 및 접속**

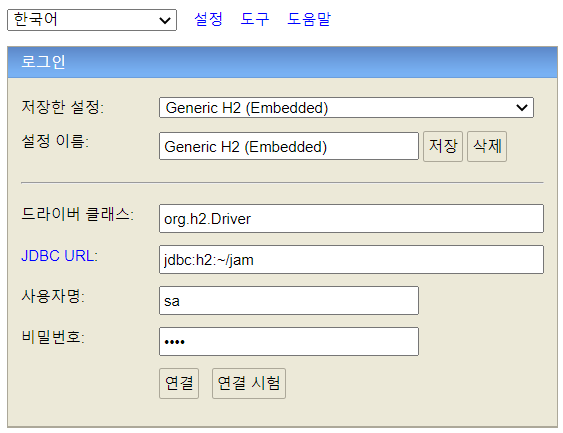
H2 데이터베이스는 교육용으로 좋은 데이터베이스로 [H2 데이터베이스 설치 사이트](https://www.h2database.com/html/download-archive.html) 에 가서 **1.4.200** 버전을 설치해준다.



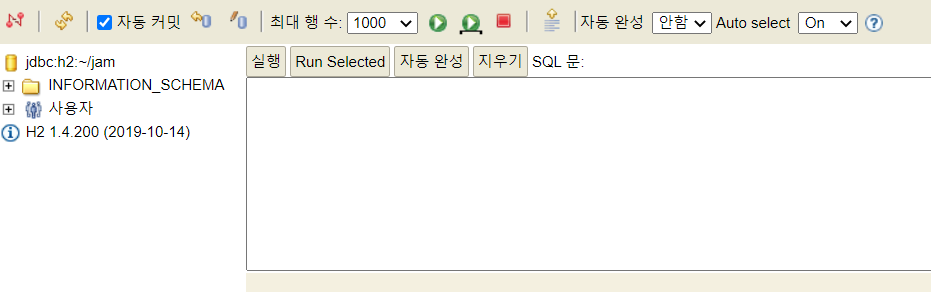


설치를 한 후, **H2 데이터베이스**를 생성해야 하기 때문에 [H2 데이터베이스 생성 방법(window)](https://www.evernote.com/shard/s568/client/snv?noteGuid=fb1f3c6b-e985-e08f-4523-a34e523406e4&noteKey=e3976a810c1854c90695f9cddf1ab7bf&sn=https%3A%2F%2Fwww.evernote.com%2Fshard%2Fs568%2Fsh%2Ffb1f3c6b-e985-e08f-4523-a34e523406e4%2Fe3976a810c1854c90695f9cddf1ab7bf&title=H2%2B%25EB%258D%25B0%25EC%259D%25B4%25ED%2584%25B0%25EB%25B2%25A0%25EC%259D%25B4%25EC%258A%25A4%2B%25EC%2583%259D%25EC%2584%25B1%2B%25EB%25B0%25A9%25EB%25B2%2595%2528%25EC%259C%2588%25EB%258F%2584%25EC%259A%25B0%2529) 을 참고해 **H2 데이터베이스**를 만든다.

만든 후, **H2 실행파일**을 실행하고 만든 **H2 데이터베이스** 경로와 사용자명, 비밀번호를 입력해주고 연결 버튼을 누른다.



그러면 **H2 데이터베이스**에 정상적으로 접속이 된다.  
추가적으로 이후에 다시 **H2 데이터베이스**에 접속할 때 **JDBC URL**를 "**jdbc:h2:~/db이름**" 이 아닌 "**jdbc:h2:tcp://localhost/~/db이름**" 로 접속해야 한다.



**🖋️ H2 데이터베이스 테이블 생성 및 실행**

**H2 데이터베이스**에 접근해서 **member 테이블** 생성한다.

* **bigint**: java로 하면 long 타입
* **generated by default as identityid**: id값으로 null이 들어오면 자동적으로 값을 채움
* **varchar(s)**: 가변 길이 문자열, s만큼 최대길이 가질 수 있음

create table member

(

id bigint generated by default as identity,

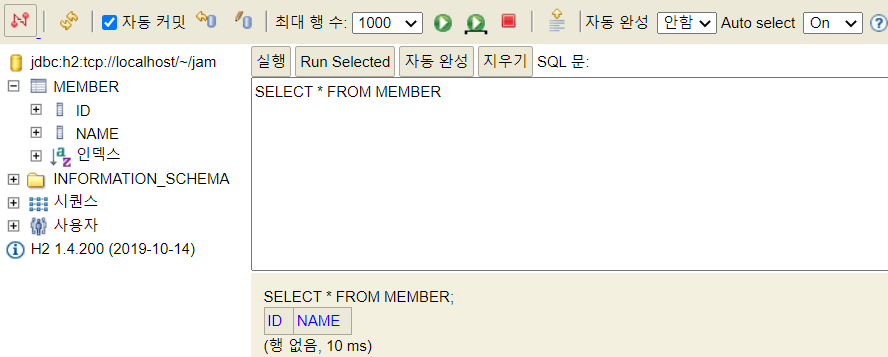
name varchar(255),

primary key (id)

);

**select** 명령어로 만든 **member 테이블**을 조회하여 확인할 수 있다.

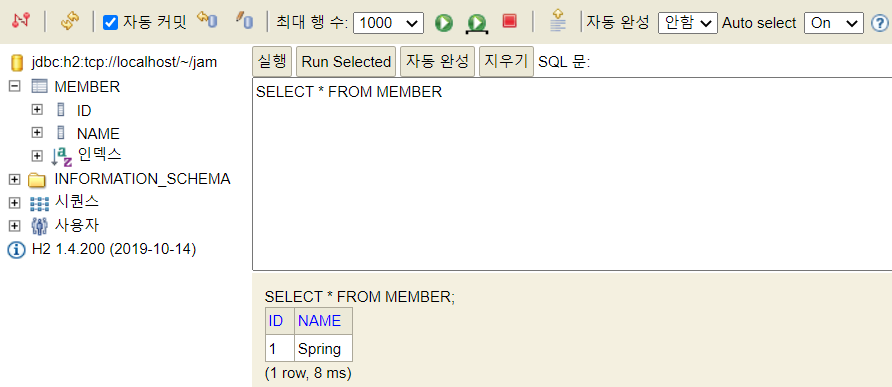
SELECT \* FROM MEMBER



**insert** 명령어로 **member 테이블**에 값을 추가할 수 있다.

INSERT INTO MEMBER(NAME) VALUES('Spring')

값을 넣은 후, **member 테이블**을 조회하면 정상적으로 데이터가 추가된 것을 확인할 수 있다.



**📒 순수 JDBC**

이번에는 직접 memory가 아닌 **DB**에 데이터를 저장하여, 넣고 빼는 것을 할 것이다.

그래서 우선 옛날에 쓰던 방식부터 시작할 것이며, 크게 중요한 부분은 아니다.

**🖋️ 순수 Jdbc 환경설정**

build.gradle에 **jdbc, h2 데이터베이스** 관련 라이브러리를 추가한다.

build.gradle

dependencies {

...

implementation 'org.springframework.boot:spring-boot-starter-jdbc'

runtimeOnly 'com.h2database:h2'

...

}

스프링 부트 데이터베이스 연결 설정 추가한다.

src/main/resources/application.properties

spring.datasource.url=jdbc:h2:tcp://localhost/~/db이름

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=비밀번호

**🖋️ Jdbc 리포지토리 생성**

src/main/java/repository/JdbcMemberRepository

package spring.study1.repository;

import spring.study1.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 spring.study1;

...

@Configuration

public class SpringConfig {

private DataSource dataSource;

@Autowired

public SpringConfig(DataSource dataSource) {

this.dataSource = dataSource;

}

...

@Bean

public MemberRepository memberRepository() {

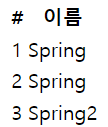
// return new MemoryMemberRepository();

return new JdbcMemberRepository(dataSource);

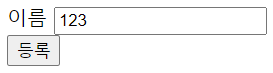
}

}

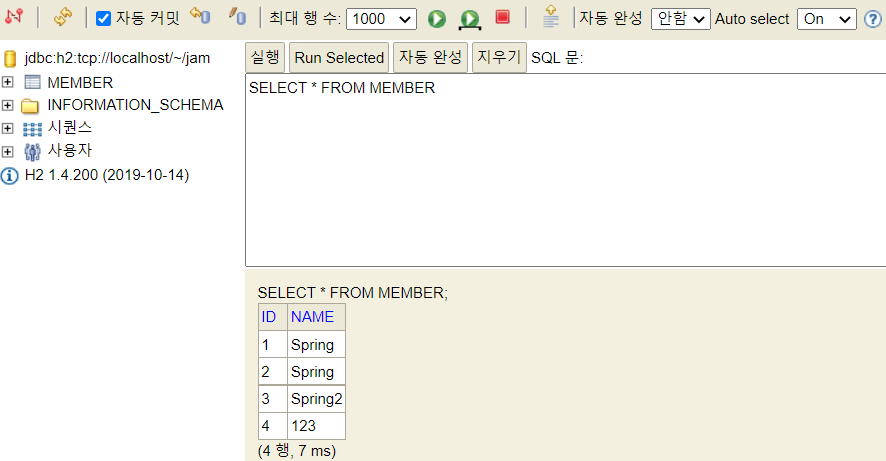
코드를 다 작성하고 실행을 하여 <http://localhost:8080/members> 에 들어가면 , **H2 DB**에서 추가한 데이터들의 목록이 보인다.



추가적으로 회원 가입 창에서 새롭게 회원을 등록하고

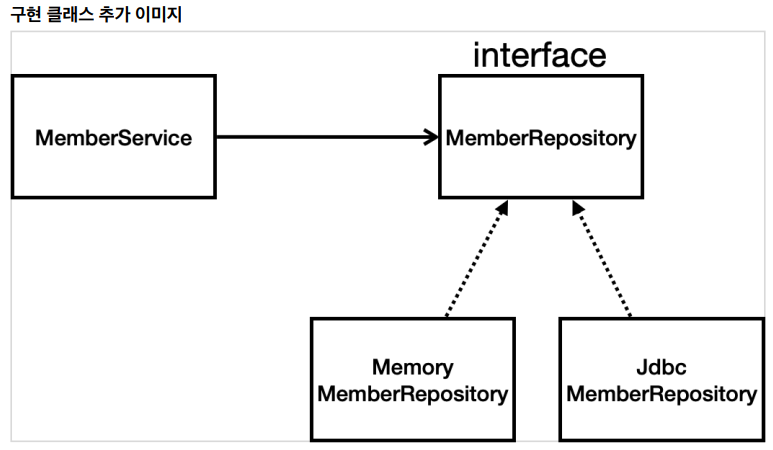


**H2 DB**에서 확인을 해보면 정상적으로 추가가 된것도 확인할 수 있다.

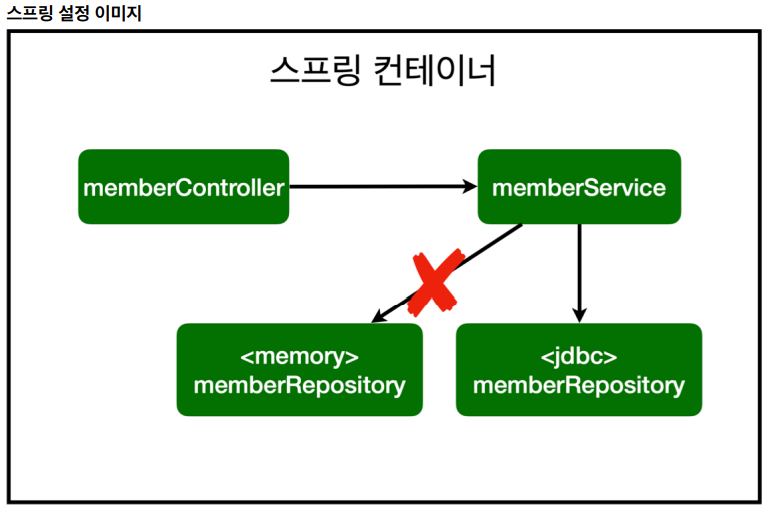


**🖋️ 설정 변경 설명**

MemberService는 MemberRepository를 의존하며, MemberRepository 구현체에는 Memory, **Jdbc MemberRepository**가 있다.



스프링 컨테이너에서 설정을 바꿔, 기존 MemoryMemberRepository을 spring bean에서 빼고 **Jdbc MemberRepository**를 새로운 spring bean으로 등록을 했다.



**개방-폐쇄 원칙(OCP, Open-Closed Principle)**  
확장에는 열려있고, 수정, 변경에는 닫혀있다.

조립하는 코드는 수정해야 하지만, 스프링의 **DI (Dependencies Injection)** 을 사용하면 기존 코드를 바꾸지 않고 설정만으로도 구현 클래스를 변경할 수 있다.

**📒 스프링 통합 테스트**

**스프링 컨테이너**와 **DB**까지 연결한 통합 테스트를 순수한 JAVA코드가 아닌 **스프링**과 엮어서 진행할 것이다.

**🖋️ 회원 서비스 스프링 통합 테스트**

MemberServiceTest의 내용을 복사붙어넣기를 해서 수정한다.

src/test/java/spring.study1.service/MemberServiceIntegrationTest

package spring.study1.service;

...

@SpringBootTest

@Transactional

class MemberServiceIntegrationTest {

@Autowired MemberService memberService;

@Autowired MemberRepository memberRepository;

@Test

void 회원가입() {

// given

Member member = new Member(); // member 객체 생성

member.setName("spring"); // member name에 hello을 넣음

// when

Long saveId = memberService.join(member); // member 객체를 회원가입하고, 반환된 id를 saveId

// then

// 회원가입한 member의 id가 저장소에 있으면, 해당 member를 findMember 로

Member findMember = memberService.findOne(saveId).get();

// 회원가입한 member와, 저장소에서 가져온 member의 이름이 같은 지 검증

Assertions.assertThat(member.getName()).isEqualTo(findMember.getName());

}

@Test

public void 중복\_회원\_예외() {

// given

// 이름이 같은 중복 회원 member 객체 생성

Member member1 = new Member();

member1.setName("spring");

Member member2 = new Member();

member2.setName("spring");

// when

memberService.join(member1);

// memberService.join(member2)에 IllegalStateException 예외 검증

IllegalStateException e = assertThrows(IllegalStateException.class, () -> memberService.join(member2));

System.out.println(e.getMessage());

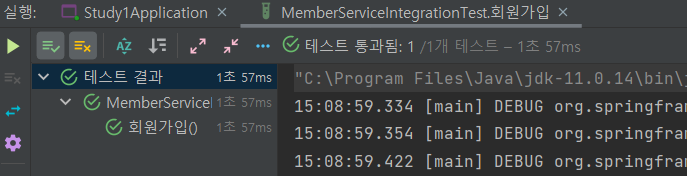
}

}

**Test**를 실행하기 앞서 **DB**에 있는 내용을 모두 지워준다.

DELETE FROM MEMBER

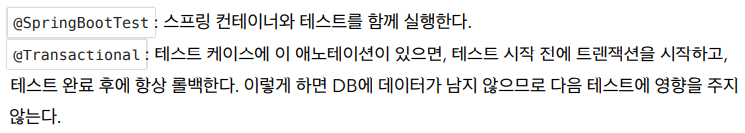
**Test**를 실행하면 **Spring**이 띄어지고 **Test**가 정상적으로 동작하는 것을 확인할 수 있다.



**🖋️ 스프링 통합 테스트 설명**

위에 코드에서 **@Transactional** 이라는 애노테이션이 있으면 **Test**를 실행할 때 트랜잭션을 먼저 실행하고, **DB**에 데이터를 넣은 후 **Test**가 끝나면 항상 **롤백**을 해준다.

그래서 쉽게 말하면 **@Transactional**가 없으면 **Test**할 때 넣었던 데이터가 **DB**에 반영이 되고, 다시 **Test**를 실행하면 **DB**에 중복된 데이터가 있어 매번 **Test**를 실행해주기 전에 **DB**에 있는 데이터를 지워야 하는 불편함이 발생한다.  
바로 이러한 번거로움을 **@Transactional**가 해결해준다.



**📒 스프링 JdbcTemplate**

**스프링 JdbcTemplate**

* 설정은 순수 Jdbc와 동일하게 환경설정
* Jdbc API에서 본 반복 코드 대부분을 제거해주지만, SQL은 직접 작성

**🖋️ 스프링 JdbcTemplate 회원 리포지토리**

src/main/java/spring.study1/repository/JdbcTemplateMemberRepository

package spring.study1.repository;

...

public class JdbcTemplateMemberRepository implements MemberRepository{

private final JdbcTemplate jdbcTemplate;

@Autowired

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 Optional<Member> findByName(String name) {

List<Member> result = jdbcTemplate.query("select \* from member where name = ?", memberRowMapper(), name);

return result.stream().findAny();

}

@Override

public List<Member> findAll() {

return jdbcTemplate.query("select \* from member", memberRowMapper());

}

private RowMapper<Member> memberRowMapper() {

return (rs, rowNum) -> {

Member member = new Member();

member.setId(rs.getLong("id"));

member.setName(rs.getString("name"));

return member;

};

}

}

작성한 **JdbcTemplate**을 사용할 수 있도록 **스프링 설정을 변경**해준다.

src/main/java/spring.study1/SpringConfig

package spring.study1;

...

@Configuration

public class SpringConfig {

...

@Bean

public MemberRepository memberRepository() {

// return new MemoryMemberRepository();

// return new JdbcMemberRepository(dataSource);

return new JdbcTemplateMemberRepository(dataSource);

}

}

코드를 다 작성한 후, 위에서 만든 **스프링 통합 테스트**를 실행하면 정상적으로 작동하는 것을 확인할 수 있다.

**📒 JPA**

**JPA**

* 기존의 반복 코드는 물론이고, 기본적인 SQL도 JPA가 직접 만들어서 실행
* SQL과 데이터 중심의 설계에서 객체 중심의 설계로 패러다임을 전환
* 개발 생산성을 크게 높임

**🖋️ 스프링 부트에 JPA 설정 추가**

build.gradle 파일에 JPA, h2 데이터베이스 관련 라이브러리를 추가한다.

build.gradle

dependencies {

...

// implementation 'org.springframework.boot:spring-boot-starter-jdbc'

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

runtimeOnly 'com.h2database:h2'

...

}

스프링 부트에 **JPA 설정**을 추가한다.

src/main/resources/application.properties

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=none

**show-sql**은 **JPA**가 생성하는 **SQL**을 출력하고,  
**ddl-auto**은 자동으로 **table**을 만들어주는 기능인데, **none**을 사용하면 해당 기능을 끈다.

**🖋️ JPA 추가**

**JPA**를 사용하기 전에, **JPA 엔티티 매핑**을 먼저 해야한다.

src/main/java/spring.study1/domain/Member

package spring.study1.domain;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity // JPA가 관리하는 엔티티

public class Member {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

...

}

본격적으로 **JPA 회원 리포지토리**를 만들어 준다.

src/main/java/spring.study1/repository/JpaMemberRepository

package spring.study1.repository;

...

public class JpaMemberRepository implements MemberRepository {

// EntityManager

private final EntityManager em;

public JpaMemberRepository(EntityManager em) {

this.em = em;

}

@Override

public Member save(Member member) {

em.persist(member);

return member;

}

@Override

public Optional<Member> findById(Long id) {

Member member = em.find(Member.class, id);

return Optional.ofNullable(member);

}

@Override

public Optional<Member> findByName(String name) {

List<Member> result = em.createQuery("select m from Member where m.name = :name",Member.class)

.setParameter("name",name)

.getResultList();

return result.stream().findAny();

}

@Override

public List<Member> findAll() {

return em.createQuery("select m from Member m", Member.class)

.getResultList();

}

}

추가적으로 **JPA**를 사용하기 위해 **트랜잭션**이라는 것이 필요해 서비스계층에 **트랜잭션**을 추가해준다.

src/main/java/spring.study1/service/MemberService

package spring.study1.service;

...

@Transactional

public class MemberService {

...

}

**🖋️ JPA 실행**

작성한 JPA를 실행하기 위해, 스프링 설정을 변경한다.

src/main/java/spring.study1/SpringConfig

package spring.study1;

...

@Configuration

public class SpringConfig {

private final DataSource dataSource;

private final EntityManager em;

public SpringConfig(DataSource dataSource, EntityManager em) {

this.dataSource = dataSource;

this.em = em;

}

...

// Spring bean에 MemberRepository 등록

@Bean

public MemberRepository memberRepository() {

// return new MemoryMemberRepository();

// return new JdbcMemberRepository(dataSource);

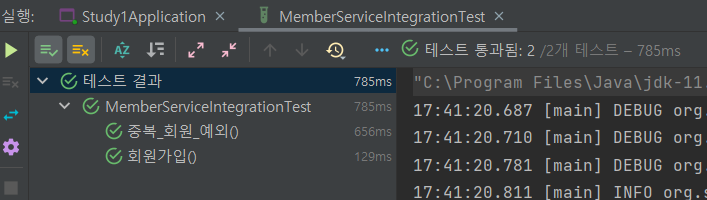
// return new JdbcTemplateMemberRepository(dataSource);

return new JpaMemberRepository(em);

}

}

정상적으로 동작하는지 확인하기 위해 **Test**를 돌려 확인을 하면, 정상적으로 동작되는 것을 볼 수 있다.



**📒 스프링 데이터 JPA**

**스프링 데이터 JPA**

* 개발 생산성이 증가하고, 개발해야할 코드도 줄어듦.
* 리포지토리에 구현 클래스 없이 인터페이스 만으로 개발
* 기본 CRUD 기능도 모두 제공

**🖋️ 스프링 데이터 JPA 회원 리포지토리**

환경 설정은 위에서 한 **JPA** 그대로 사용하며, **스프링 데이터 JPA 회원 리포지토리**를 자바 클래스가 아닌 **인터페이스**로 만들어 준다.

src/main/java/spring.study1/repository/SpringDataJpaMemberRepository

package spring.study1.repository;

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

import spring.study1.domain.Member;

import java.util.Optional;

// 스프링 데이터 JPA가 SpringDataJpaMemberRepository를 스프링 빈으로 자동 등록

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

@Override

Optional<Member> findByName(String name);

}

그리고 **스프링 데이터 JPA 회원 리포지토리**를 사용하도록 스프링 설정을 변경한다.

src/main/java/spring.study1/SpringConfig

package spring.study1;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import spring.study1.repository.\*;

import spring.study1.service.MemberService;

@Configuration

public class SpringConfig {

private final MemberRepository memberRepository;

@Autowired

public SpringConfig(MemberRepository memberRepository) {

this.memberRepository = memberRepository;

}

@Bean

public MemberService memberService() {

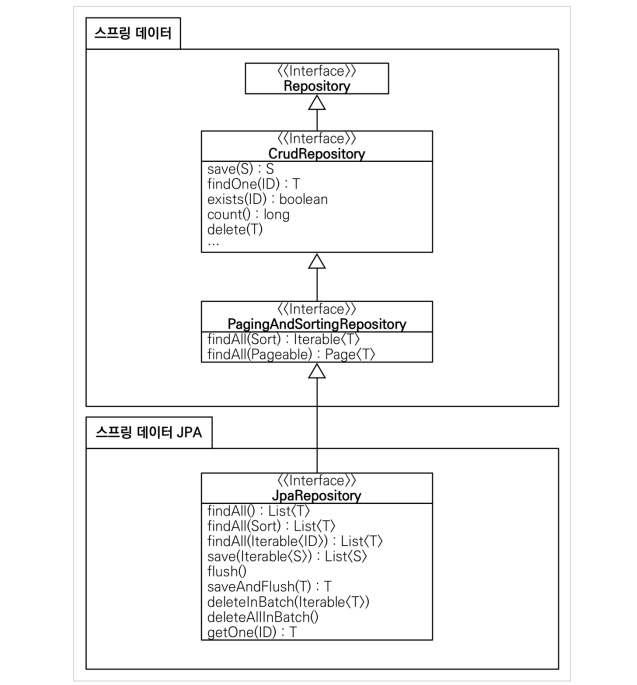
return new MemberService(memberRepository);

}

}

**스프링 데이터 JPA 회원 리포지토리**가 정상적으로 동작하는지 확인하기 위해 Test를 돌려 실행하면, 정상적으로 동작하는 것을 확인할 수 있다.

**🖋️ 스프링 데이터 JPA 설명**



**스프링 데이터 JPA**는 인터페이스를 통한 기본적인 CRUD, findByName() , findByEmail() 처럼 메서드 이름 만으로 조회 기능, 페이징 기능들을 제공한다.