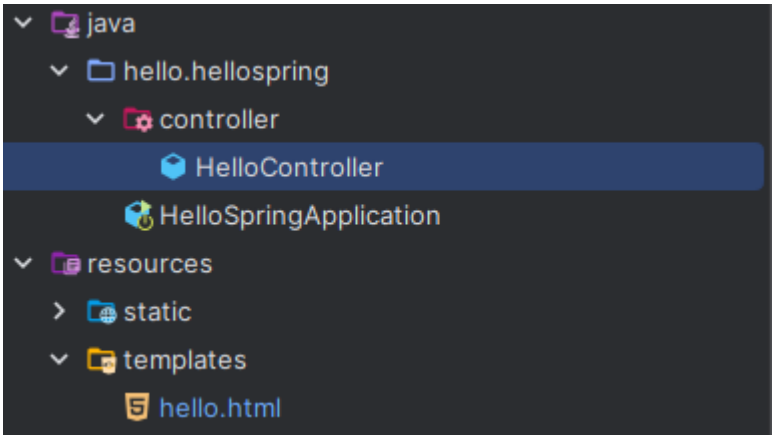
스프링 입문

▼ Section1. 프로젝트 설정

프로젝트 생성

- Java 17
- Gradle
- Dependencies
  - Spring web
  - Thymeleaf



HelloController, hello.html 실습 코드 진행

▼ Section2. 스프링 웹 개발 기초

정적 컨텐츠

- 서버 처리 X, 파일(원본)을 내려줌

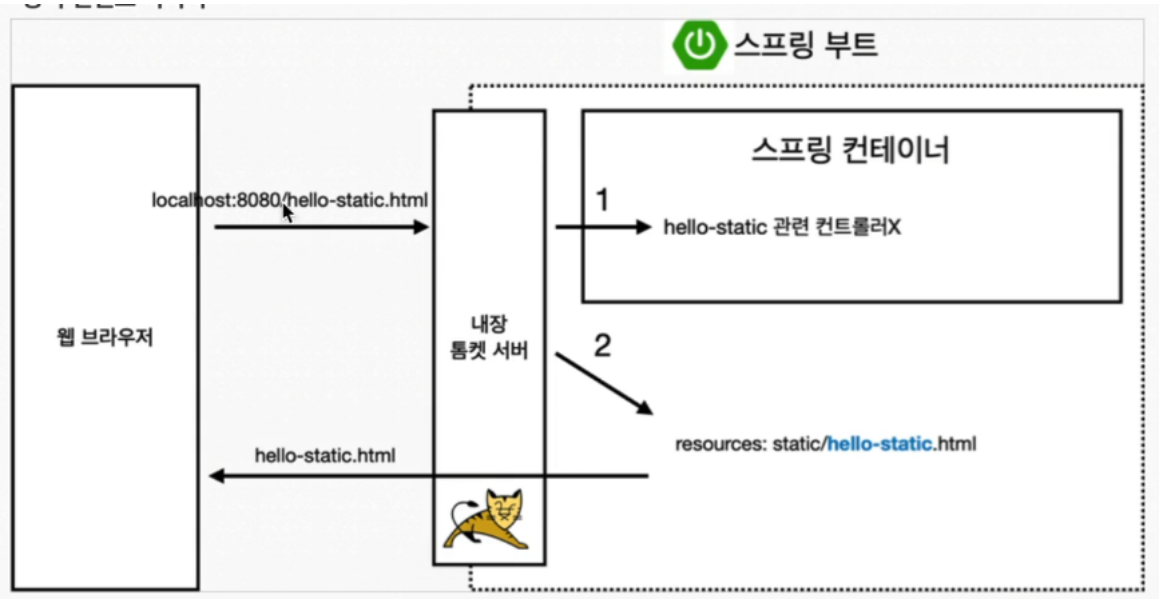
`MVC와 템플릿 엔진`

- 서버에서 HTML 과 같은 정적 컨텐츠를 동적으로 처리하여 내려줌

`API`

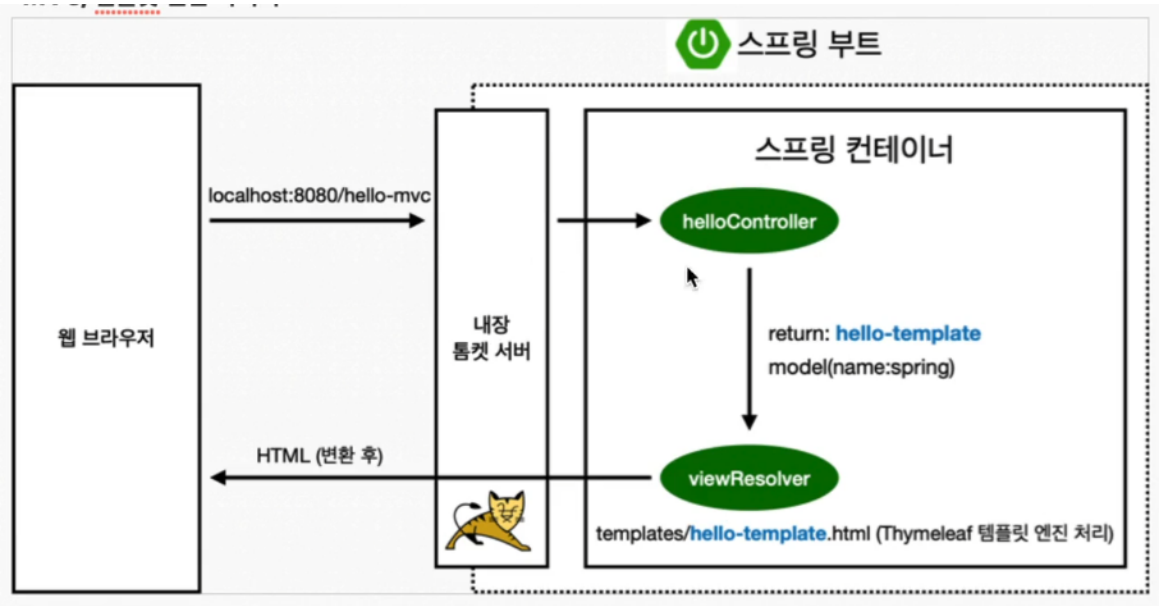
- 과거에는 **xml**, 현재에는 주로 **json** data formmat 으로 데이터 전달

정적 컨텐츠



- [resources] - [static] html 파일 생성
- View 과정
  1. 내장 톰캣 서버가 요청을 받고 스프링에게 넘기면,
  2. 스프링은 hello-static 컨트롤러를 찾아봄
  3. resources 안에 있는 hello-static을 찾아가서 반환

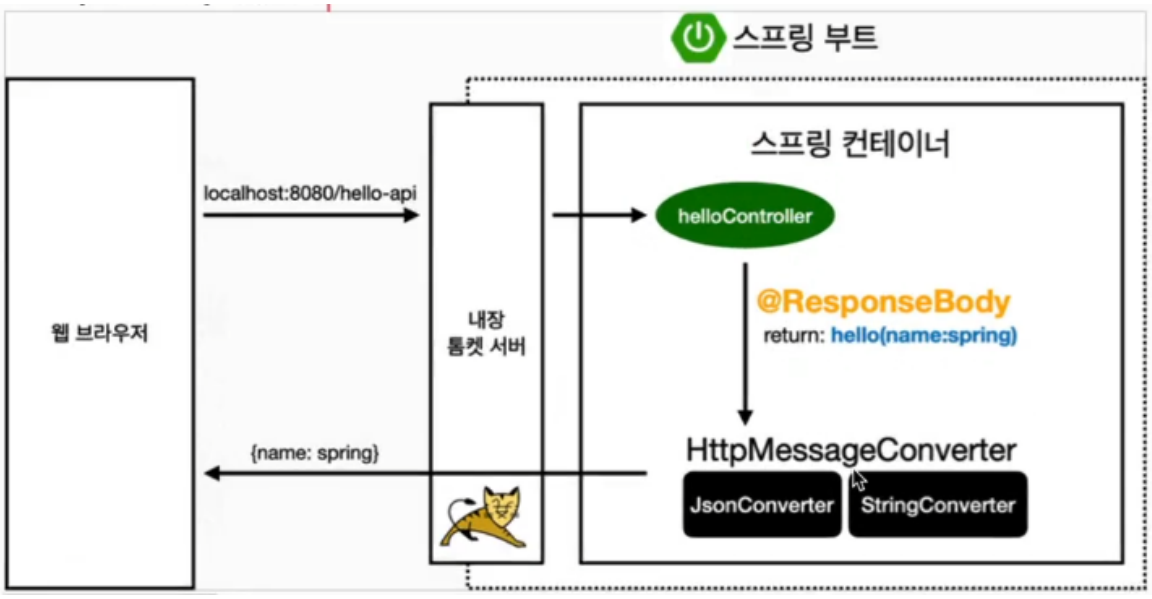
MVC와 템플릿 엔진




```
// HelloController 에서 다음과 같은 코드 작성
@GetMapping("hello-mvc")
public String nonHello(Model model){ // method명은 hello, nc
    model.addAttribute("data", "spring");

    // client 요청: localhost:8080/hello-mvc
    // hello 로 반환하여 톰캣이 client의 요청을 받으면
    // spring은 컨테이너 안의 controller 중 hello-mvc 관련 cont
    // viewResolver를 통해 template 폴더 안에서 hello-mvc.html

    // 관련 controller 가 없다면 static 폴더 안의 hello-mvc를 찾
    // static 에도 hello-mvc가 없다면 error page 로 이동
    return "hello-mvc";
}
```

- `requestParam`: url을 통한 param 받기 (value = “이름”), required 는 default: true 생략가능

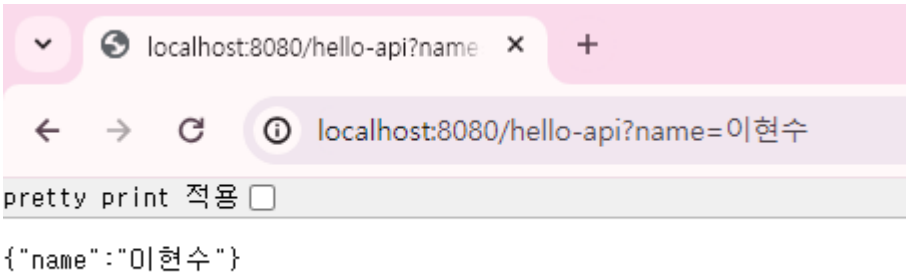
API



- 웹 브라우저에서 `localhost:8080/hello-api` 요청을 보낼 때 기본 반환 정책
  - controller 의 return이 String 인 경우 String 그대로 http 응답에 반환
  - return 이 **Object 인 경우 json 형태로 만든 후** http 응답에 반환
    - viewResolver가 아닌 `HttpMessageConverter` 가 동작

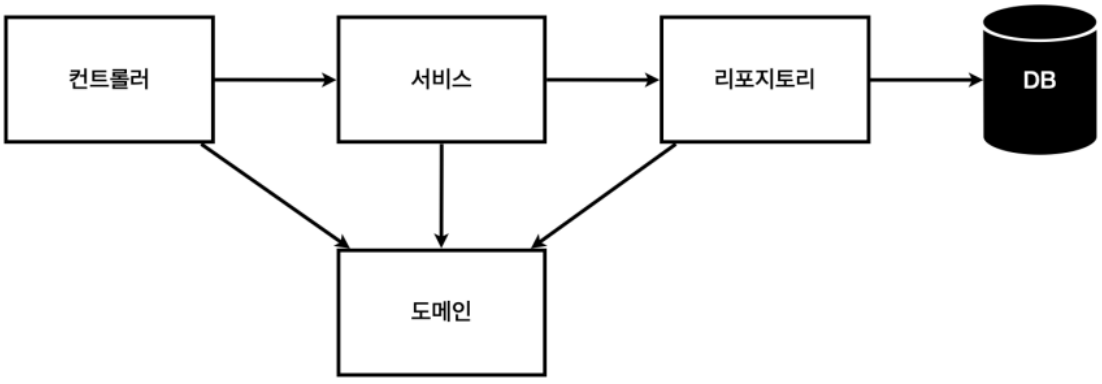
```
@GetMapping("hello-api")
@ResponseBody // http body 부분에 아래 데이터를 직접 넣어주겠다.
public Hello helloApi(@RequestParam(value = "name") String
    Hello hello = new Hello();
        hello.setName(name);
    return hello;
}

static class Hello{
    String name;

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

- 템플릿과의 차이는 view 없이 문자 그대로 내려준다.
- 문자열뿐만 아니라 객체(Object) 전달 가능
    - localhost:8080/hello-api?name="이현수" 요청 시



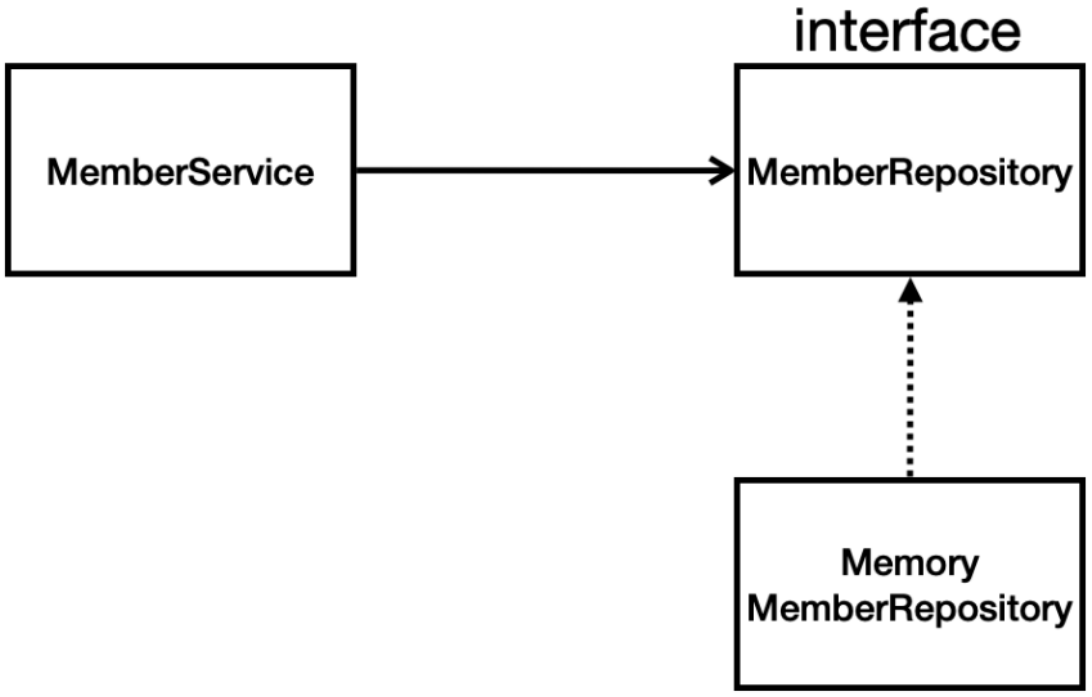
▼ Section3. 회원 관리 예제 - BackEnd

일반적인 웹 애플리케이션 계층 구조



- 컨트롤러: 웹 MVC의 컨트롤러 역할
- 서비스: 핵심 비즈니스 로직 구현
- 리포지토리: 데이터베이스에 접근, 도메인 객체를 DB에 저장하고 관리
- 도메인: 비즈니스 도메인 객체, 예) 회원, 주문, 쿠폰 등등 주로 데이터베이스에 저장하고 관리됨

비즈니스 요구사항



- 아직 데이터 저장소가 선정되지 않아서, 우선 인터페이스로 구현 클래스를 변경할 수 있도록 설계
- 데이터 저장소는 RDB, NoSQL 등등 다양한 저장소를 고민중인 상황으로 가정
- 개발을 진행하기 위해서 초기 개발 단계에서는 구현체로 가벼운 메모리 기반의 데이터 저장소 사용

회원 도메인, 레포지토리 만들기

> 💡 실습 코드 작성

- 인터페이스로 `memberRepository` 만드는 이유
  - 추후 DB를 RDB 로 사용할 지, NoSQL 로 사용할지, 메모리 DB를 사용할 지 정해지지 않음 → 유연성

회원 레포지토리 테스트 케이스 작성

```
package hello.hellospring.repository;

import hello.hellospring.domain.Member;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class MemoryMemberRepositoryTest {
```

```
    MemberRepository repository = new MemoryMemberRepositor

    @Test
    public void save(){
        Member member = new Member();
        member.setName("spring");

        repository.save(member);

        Member result = repository.findById(member.getId())
        System.out.println("result = " + (result == member)
        Assertions.assertEquals(member, null);
    }
}
```

- `Assertions.assertEquals` 사용 시 왼쪽 인자는 기댓값, 오른쪽은 실제값을 두고
  - 같은 결과일 경우 초록색 v
  - 다른 결과일 경우 오류 출력
- `org.assertj.core.api.Assertions.` *`assertThat`* `(member).isEqualTo(result);`
  - 위 `Assertions.assertEquals` 와 똑같은 역할이지만 영한님이 더 자주 씀
  - 코드가 기니 static import 해주면 좋음
- Test 클래스 run 시 **전체 테스트**
- 메소드 run 시 **단위 테스트**

💡 테스트의 경우 순서를 보장하지 않음

`findAll()`, `findById()` 모두 테스트에서 정의 되어있을 때 member1, member2 가 이미 findAll() 에서 정의가 되어있으므로 member1, member2 를 초기화해주는 코드가 포함된 findById()의 경우 오류 발생

→
테스트가 끝나는 경우 데이터 clear 해주기

AfterEach annotation 활용하여 각 테스트가 끝나는 각각의 경우 clear() 메서드 실행

회원 서비스 개발

💡 실습 코드 작성

회원 서비스 테스트

- `ctrl + shift + T` 테스트케이스 생성 단축키

테스트 진행 시 다음 구조를 따르면 좋다.

1. given
2. when
3. then

💡 실습 코드 작성

```
MemberService memberService = new MemberService();
MemoryMemberRepository memberRepository = new MemoryMemberR

// 위 코드의 경우 테스트 실행 시 마다 repo 와 service가 따로 논다.
// 이 경우에는 같은 repo 를 쓰기 위해 아래와 같이 바꿔주고
// MemberService 코드에서는 생성자의 파라미터를 통해 memberReposit
```

```
// 같은 repo를 쓰게 한다.

// 이러한 과정을 DI(Dependency Injection) 이라고 한다.
MemberService memberService;
MemoryMemberRepository memberRepository;

@BeforeEach
public void beforeEach(){
    memberRepository = new MemoryMemberRepository();
    memberService = new MemberService(memberRepository);
}
```

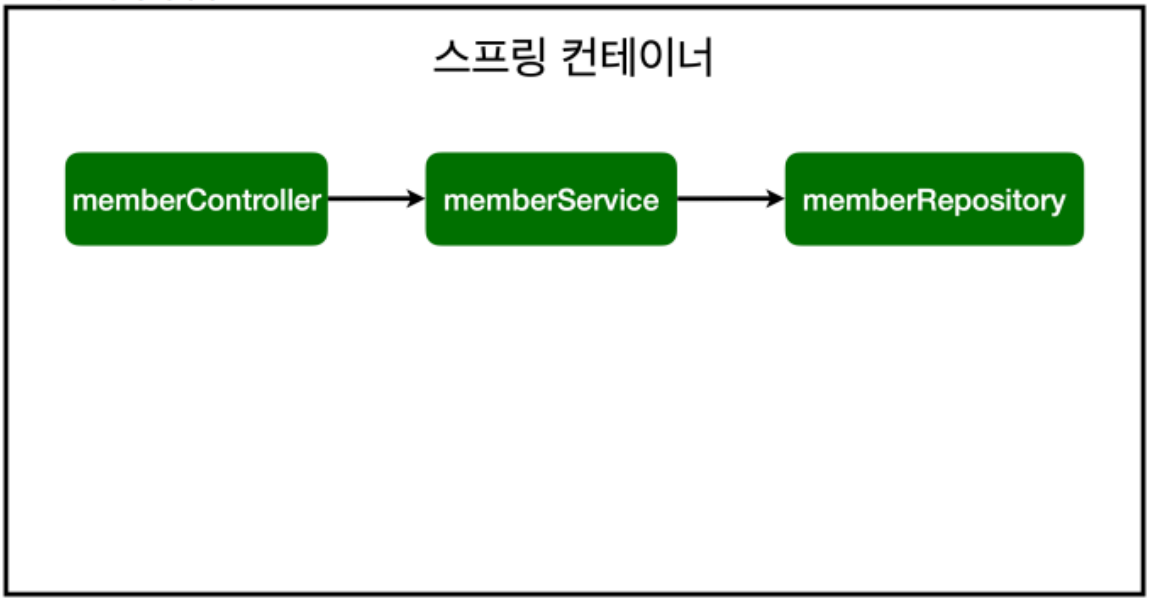
▼ Section4. 스프링 빈과 의존관계

스프링 빈을 등록하는 2가지 방법

- 컴포넌트 스캔과 자동 의존관계 설정
- 자바 코드로 직접 스프링 빈 등록하기

컴포넌트 스캔과 자동 의존관계 설정

스프링 빈 등록 이미지



- 동일 패키지, 하위 패키지이 대상이 된다.
    - 설정을 통해 다른 패키지도 가능하다.

`@Controller` 어노테이션이 붙어있는 경우 객체가 생성되고, 스프링이 해당 객체를 가지고 있음

⇒ 이를 '스프링 컨테이너에서 스프링 빈이 관리된다' 라고 표현

`@Autowired` 어노테이션을 쓰는 경우 스프링 컨테이너에 등록된 빈(객체)을 끌고온다.

⇒ **Dependency Injection(DI)** 의존관계 주입

> 💡 `@Component` ⊃ `@Service`, `@Repository` 또한 스프링 컨테이너에 등록해주는 어노테이션이다.
> Service, Repository의 경우 Component 가 붙어 있음
>
> 컨테이너에 등록되면 기본적으로 싱글톤으로 등록된다.

자바 코드로 직접 스프링 빈 등록하기

- 직접 Configuration 클래스를 만들어 Bean 등록

```
package hello.hellospring;

@Configuration
public class SpringConfig {

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

    @Bean
    public MemberRepository memberRepository(){
        return new MemoryMemberRepository();
    }
}
```

▼ Dependency Injection

생성자 주입(권장)

- 위 예제에 경우 생성자를 통해서 `memberRepository` 가 들어옴
- 컨테이너가 생성되는 시점 딱 한번만 생성

필드 주입(권장 X)

- `@Autowired private final MemberService memberService;`

Setter 주입

```
private MemberService memberService;

@Autowired
public void setMemberService(MemberService memberService
    this.memberService = memberService;
}
```

- 단점
  - 누군가가 setMemberService() 를 호출해야하기 때문에 `public` 으로 노출되어야 함

> 💡 실무에서는 주로 정형화된 컨트롤러, 서비스, 레포지토리 같은 코드는 컴포넌트 스캔 사용
>
> 정형화 되지 않거나, 구현 클래스를 변경해야 하면 설정을 통해 스프링 빈으로 등록 ← 요구사항의 DB가 아직 선정되지 않은 것처럼

▼ Section5. 회원 관리 예제 - 웹 MVC

회원 웹 기능 - 홈 화면 추가

> 💡 실습 코드 작성

- 정적 리소스 보다 template 리소스의 우선순위가 더 높다.

회원 웹 기능 - 등록

💡 실습 코드 작성

- Form 태그의 action = “post”, controller 의 `@PostMapping` 이 반응
- Form 태그의 input 태그의 name = “name” 이라고 되어있기 때문에
  - 스프링이 알아서 MemberForm.setName() 을 호출 → 이름 넣음

회원 웹 기능 - 조회

💡 실습 코드 작성

- jsp의 el, jstl이 떠오르는 강의 굳이 thymeleaf 까지? 느낌임

▼ Section6. 스프링 DB 접근 기술

H2 데이터베이스 설치

💡 그냥 설치하면 된다.

```
drop table if exists member CASCADE;
create table member
(
 id bigint generated by default as identity,
 name varchar(255),
 primary key (id)
);
```

순수 JDBC

💡 JDBC 는 커리큘럼에서 많이 다뤘으니 정리는 따로 안하고 실습만 진행하겠습니다.

- H2 Database 연동

```
// Build.gradle
implementation 'org.springframework.boot:spring-boot-starte
runtimeOnly 'com.h2database:h2'

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

// 구현체에서 dataSource 사용하기

// SpringConfig.java
private final DataSource dataSource;

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

// JdbcMemberRepository.java
private final DataSource dataSource;
public JdbcMemberRepository(DataSource dataSource) {
    this.dataSource = dataSource;
}
```

스프링 통합 테스트

- `@SpringBootTest`
  - 스프링 컨테이너와 테스트를 함께 실행한다.
- `@Transactional`
  - 테스트는 기본적으로 반복할 수 있어야 한다.
  - 해당 Annotation을 테스트 코드에 붙이면 테스트로 들어간 데이터를 롤백 시켜 준다.

💡 스프링 컨테이너 없이 테스트하는 순수한 단위 테스트가 더 좋다고 볼 수 있다.
→ 테스트 설계의 오류 발생할 수 있기 때문에

스프링 JdbcTemplate

- JDBC API에서 반복 코드를 대부분 제거해준다.
- SQL은 직접 작성해야 한다.

```
private JdbcTemplate jdbcTemplate;
@Autowired // 생성자가 하나라면 Autowired 생략 가능
public JdbcTemplateMemberRepository(DataSource dataSource)
    jdbcTemplate = new JdbcTemplate(dataSource);
}

--- 중략 ---

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

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

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

JPA

- 기존의 반복 코드는 물론이고 기본적인 SQL도 JPA가 직접 만들어서 실행
- SQL과 데이터 중심의 설계에서 객체 중심의 설계로 패러다임 전환
- 개발 생산성을 크게 향상 시킨다.

- JPA를 사용할 때는 항상 `@Transactional` Annotation 이 필요!!
- 예제 프로젝트에서는 `Service` 에 추가 (Service.join()에 넣어도 된다)

```
// build.gradle 추가
implementation 'org.springframework.boot:spring-boot-starte

// application.properties 추가
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none
```

```
// Member.java 추가

// Id Annotation 은 PK, GeneratedValue 은 DB에서 알아서 생성해준
@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;


// JpaMemberRepository 추가

// SpringBoot 가 build.gradle 에서 jap 를 읽어들이면 자동으로 Ent
// 우리는 이 em 을 injection 받으면 된다.
private final EntityManager em; // jpa는 em으로 모든 것(DB 통신

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

@Override
public Member save(Member member) {
```

```
    em.persist(member); // jpa 가 알아서 insert 해주고 setId ㅎ
    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 Mem
            .setParameter("name", name)
            .getResultList();

    return result.stream().findAny();
}

@Override
public List<Member> findAll() {
    // select m -> Member 자체를 select
    return em.createQuery("select m from Member m", Member.
}
```

스프링 데이터 JPA

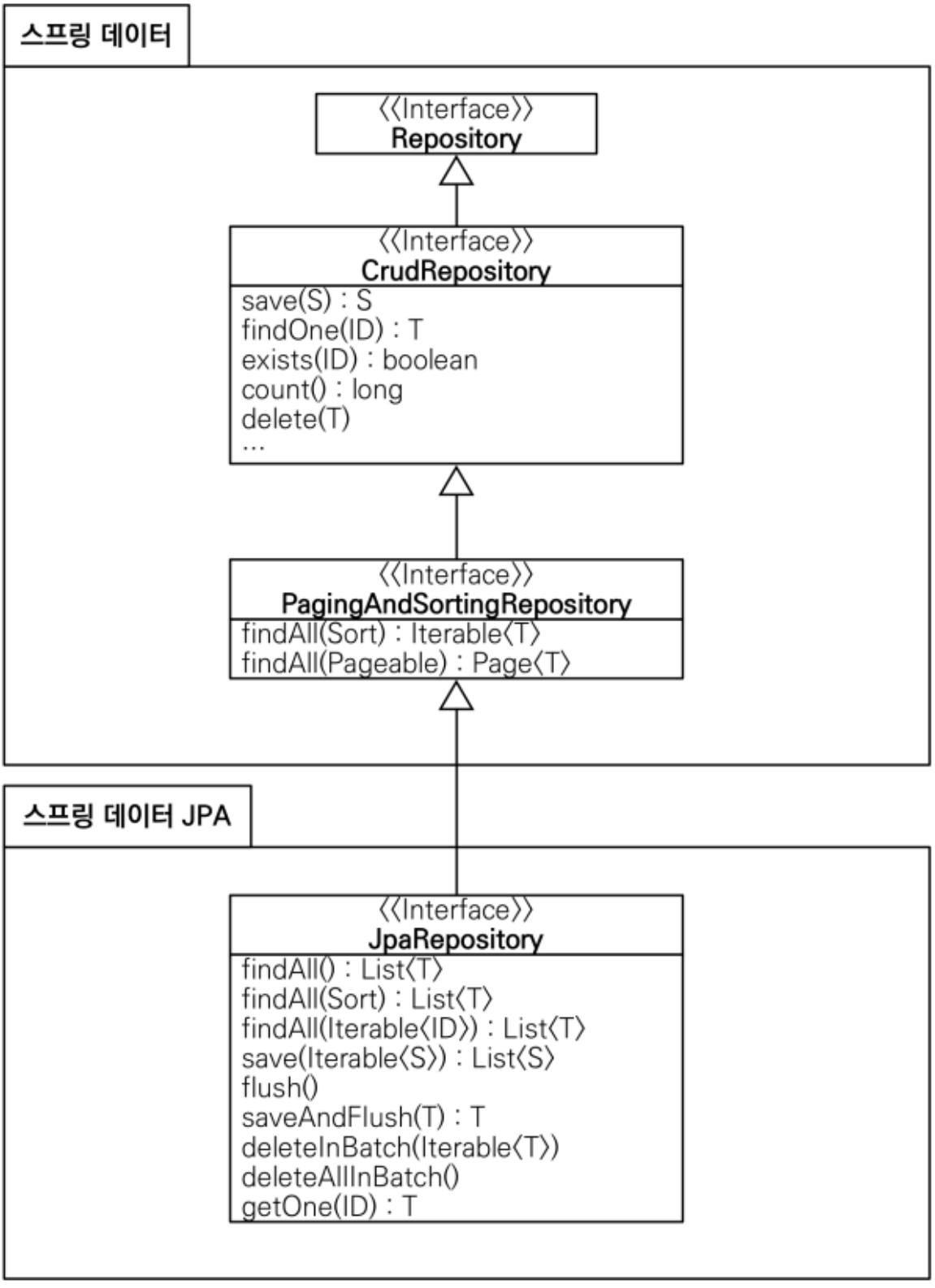
- 기존의 한계를 넘어 레포지토리에 구현 클래스 없이 인터페이스 만으로 개발이 가능해진다.
- 반복 개발해온 기본 CRUD 기능도 스프링 데이터 JPA가 모두 제공한다.

💡 JPA 설정 그대로 사용

이전 코드 그대로 + 아래 코드를 작성하니 테스트 시 오류가 뜸
오류 코드를 보니 이전에
`MemberRepository` 인터페이스를 구현한 구현체들이 `@Repository` 로 선언되어 있다보니 스프링 컨테이너에서 충돌이 나는 것 같음

Repository Annotation을 주석처리하니 해결!

```
// SpringConfig 설정
@Configuration
public class SpringConfig {
    private final MemberRepository memberRepository;
    // Spring Data Jpa 가 만들어 놓은 구현체가 자동으로 등록

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

}

// 다음 코드 적으면 findByName 구현 끝!!
public interface SpringDataJpaMemberRepository extends JpaR
    @Override
    Optional<Member> findByName(String name);
}
```



- `extends JpaRepository` 를 한 이유
  - `JpaRepository` 에 이미 기본적인 CRUD 가 구현이 되어있다.
    - 강의를 보면 JpaRepository의 시그니처 name을 그대로 따라했음

- 아무리 개발자가 공통화를 해도 `findByName()` 과 같이 **비즈니스 마다 다른 로직은 직접 인터페이스에서 네이밍 조합** 해줘야 한다.
  - findBy + Name
  - findBy + Name + [And | Or] + Id
  - 등등

> 💡 실무에서는 JPA, 스프링 데이터 JPA 사용
> 복잡한 동적 쿼리는 Querydsl 라이브러리 사용
> + → 해결하기 어려운 쿼리라면 JPA가 제공하는 네이티브 쿼리 or JdbcTemplate 사용

▼ Section7. AOP

AOP가 필요한 상황

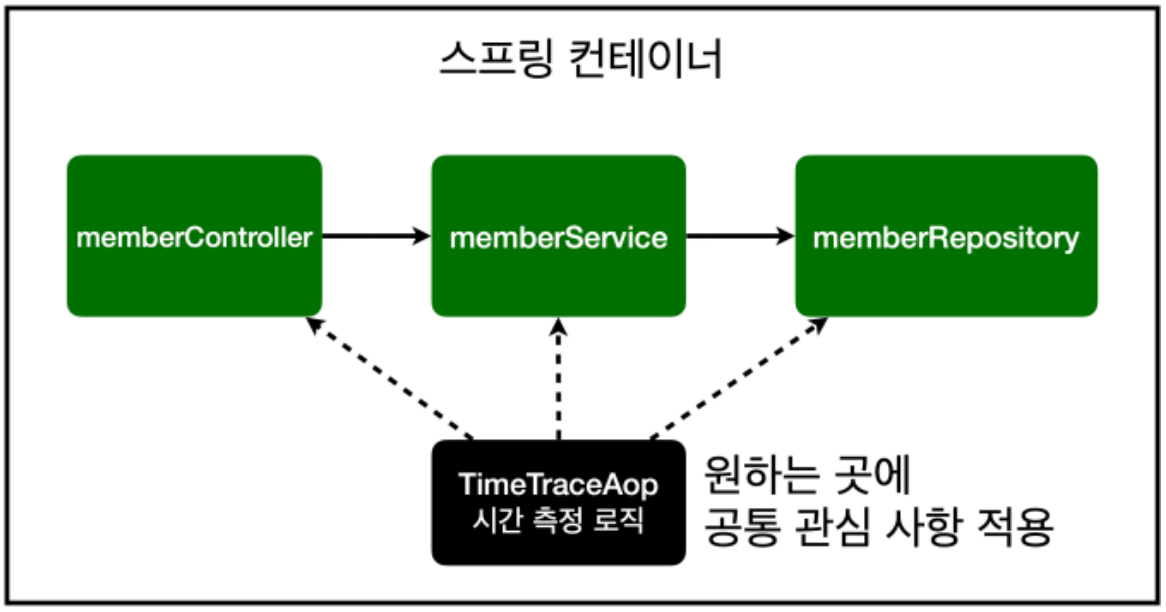
- 모든 메서드의 호출 시간 측정
- 공통 관심 사항 vs 핵심 관심 사항
- 예시로 회원 가입 시간, 회원 조회 시간을 측정

```
void join() {
    Member member = new Member();
    member.setName("spring");

    long start = System.currentTimeMillis();

    try{
        Long saveId = memberService.join(member);
        Member findMember = memberService.findOne(saveId).g
        assertThat(member.getName()).isEqualTo(findMember.g
    } finally {
        long end = System.currentTimeMillis();
        System.out.println(end - start + "ms");
    }
}
```

위 코드와 같이 시간 측정 로직, 핵심 비즈니스 로직이 섞여 있어 가독성뿐만 아니라 코드의 유지보수성도 안 좋아짐

- 공통 관심 사항: 시간 측정 로직
- 핵심 관심 사항: 비즈니스 로직

AOP(Aspect Oriented Programming) 적용

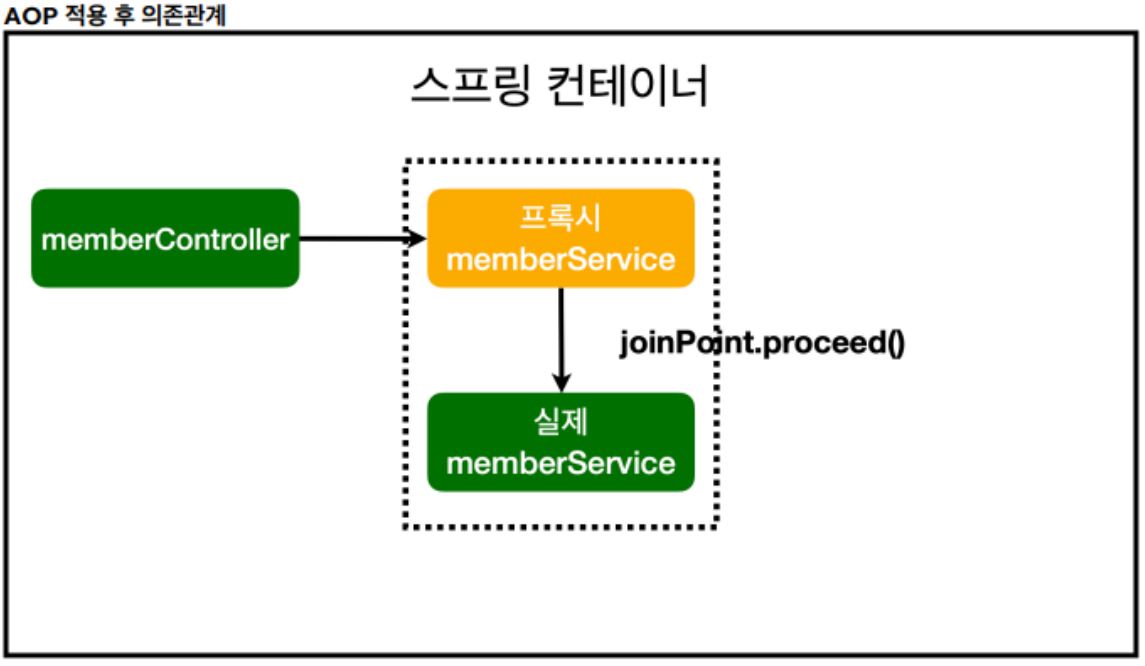


- 공통 관심 사항, 핵심 관심 사항 **분리**

예제 코드

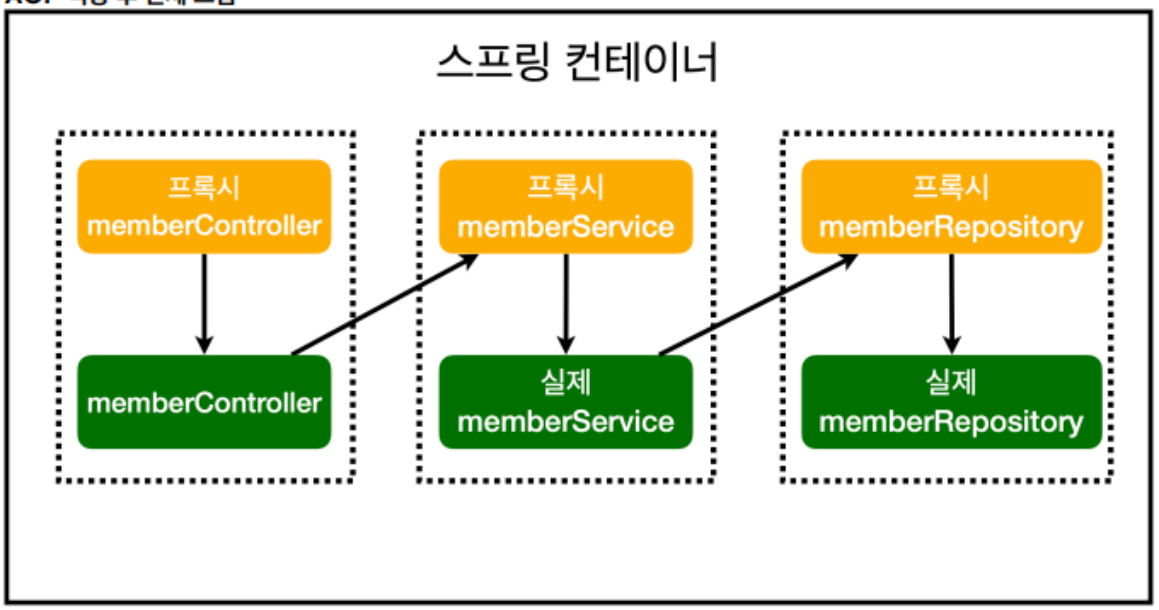
```
package hello.hellospring.aop;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

// 컴포넌트 스캔으로 등록하는 것보다 직접 Configuration에 Bean으로 
@Aspect
public class TimeTraceAop {

    @Around("execution(* hello.hellospring..*(..))") // 공통
    public Object execute(ProceedingJoinPoint joinPoint) th
```

```
        long start = System.currentTimeMillis();

        System.out.println("START: " + joinPoint.toString()
        try{
            Object result = joinPoint.proceed();// 다음 메서
        } finally {
            long finish = System.currentTimeMillis();
            long timeMs = finish - start;
            System.out.println("END: " + joinPoint.toStrin
        }
    }
}
```



AOP를 적용하면, 실제 Service가 아닌 같은 기능을 가진 가짜 Service를 실행시키고 실행이 완료되면 실제 Service 호출

AOP 적용 후 전체 그림


스프링 컨테이너
프록시
memberController
프록시
memberService
프록시
memberRepository
memberController
실제
memberService
실제
memberRepository