

YC Tech

웹 백엔드 실무 개발 프로젝트



연세대학교 미래교육원
YC (Yonsei X Codepresso) Tech Academy

1주차 recap



Controller 스펙 구현

 SNS (twitter / instagram) 의 Post 라는 도메인의 controller 를 구현

1. Post 작성 기능
2. Post 전체 조회 기능
3. Post 상세 조회 기능

Swagger 문서 작성 (optional)

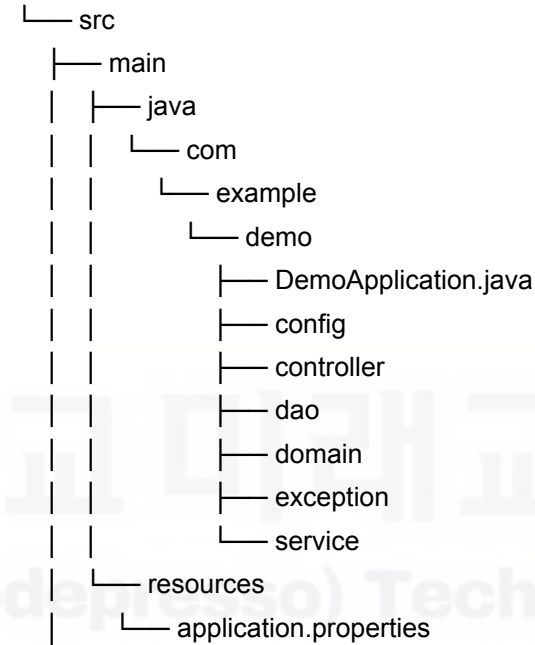
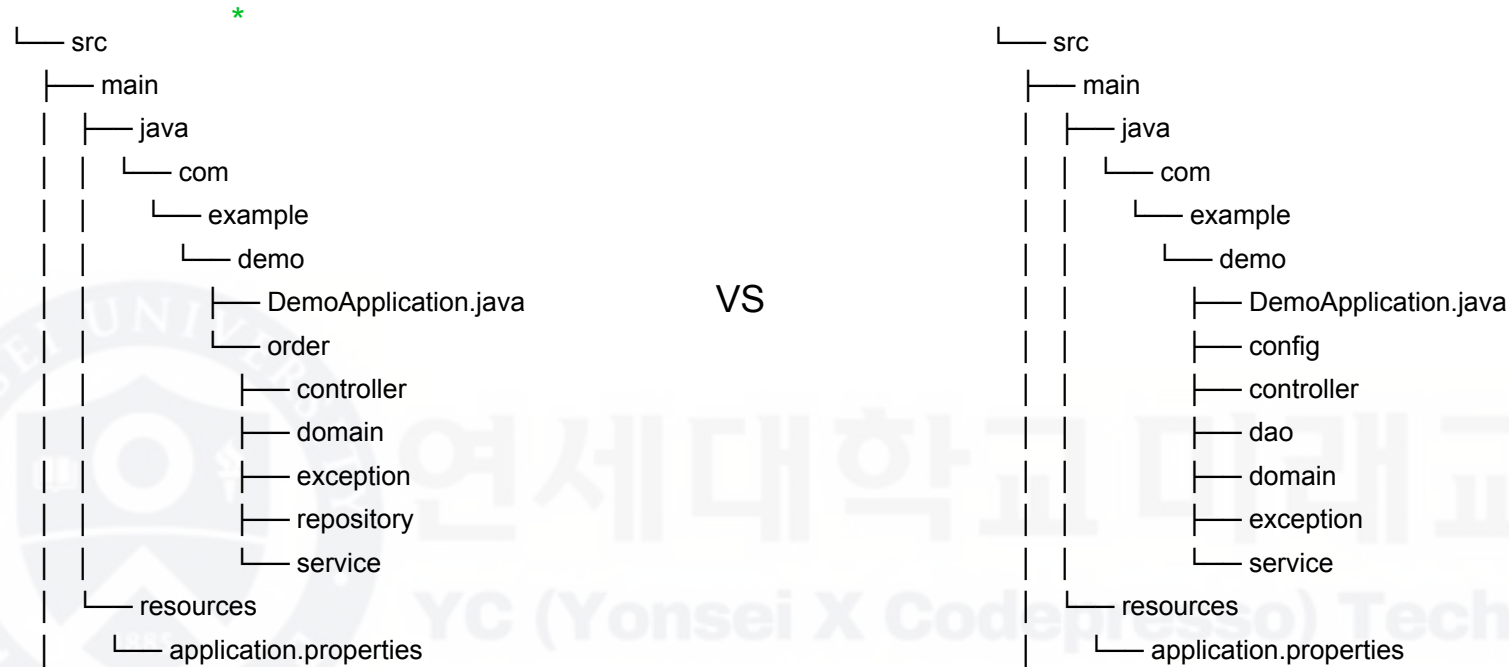
: <https://docs.sysout.co.kr/web/back-end/openapi-swagger/springdoc>



YC (Yonsei X Codepresso) Tech Academy

Spring 폴더 구조

기능(도메인) 별 vs 레이어 별 폴더 구조



- 도메인 별 구조
 - 도메인 디렉터리 기준으로 코드를 구성한다.
 - 프로젝트가 커질수록 해당 도메인에 대한 이해가 쉬워질 수 있음
- 계층 별 구조
 - 각 계층을 대표하는 디렉터를 기준으로 코드들이 구성한다.

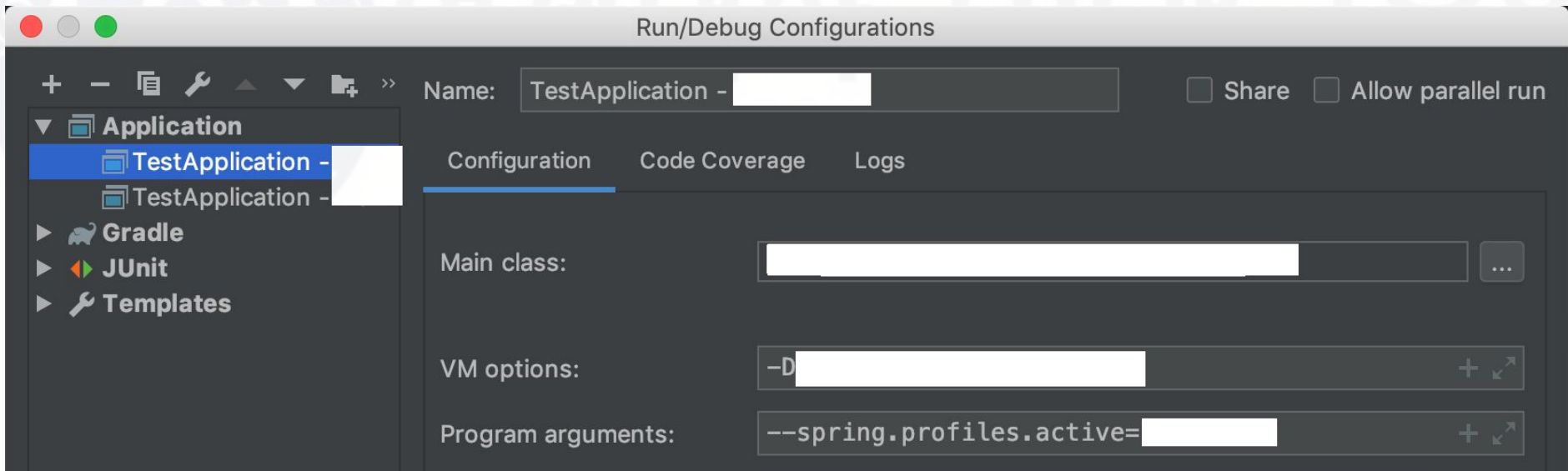
Spring profile

Profiles 을 사용하면 애플리케이션이 실행되는 환경에 따라 다른 **Bean** 들을 매핑할 수 있다.

예를 들어, 개발 환경, 스테이징 환경, 혹은 실 서비스 환경에 따라 다른 의존성을 주입할 수 있다.

설정법

- Profiles별 application-{profile}.properties 작성 (ex : application-dev.properties)
- spring.profiles.active property를 통해 default active profile을 설정 할 수 있음.



데이터 전송 규약

DTO (Data Transfer Object)

- 계층 간 데이터 교환을 하기 위해 사용하는 객체로, **DTO**는 로직을 가지지 않는 순수한 데이터 객체
 - o **Java 16**의 **record**로 **immutable**
 - o **final** 클래스이므로 다른 클래스가 상속할 수 없음

```
public record TestRecordDto(Long id, String name, String email) {}
```

데이터 전송 규약

VO (value object)

@Getter

```
public class PostsVO {
```

```
    private Long id;
```

```
    private String title;
```

```
    private String content;
```

```
    private String author;
```

@Override

```
    public boolean equals(Object o) { }
```

@Override

```
    public int hashCode() { }
```

- 변하지 않는 값을 가지는 객체(불변성, **immutable**) 값이 변하지 않음을 보장
- 값이 같다면 동일한 객체 각, 같은 객체인지 판단하기 위해 각 속성들의 값을 비교함 **equals()** 메서드와 **hashCode()** 메서드를 오버라이드해서 객체 비교를 구현

* database

가

VO

Bean



Bean

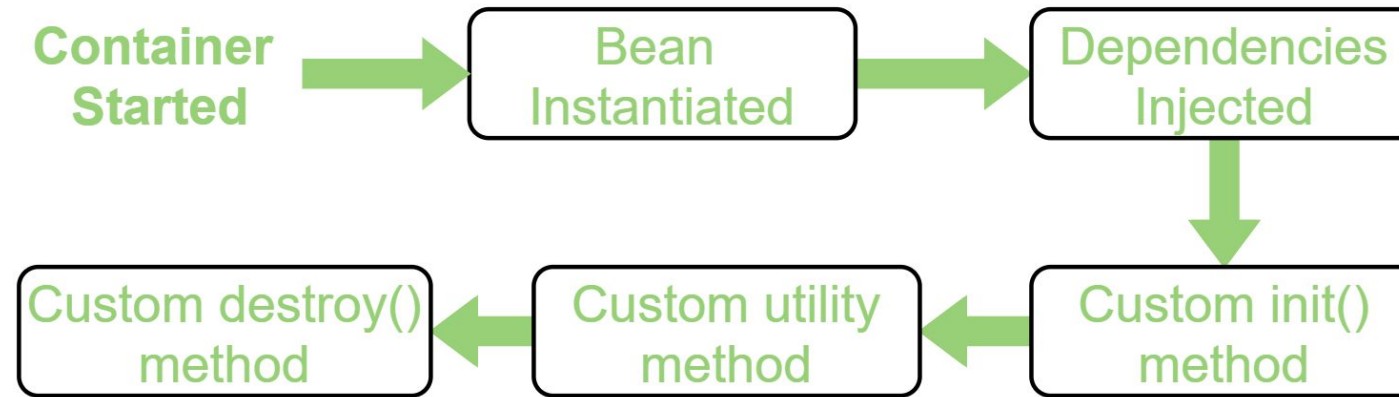
Spring container에 의해 관리되는 자바 객체(POJO)

- **@Controller** : 스프링 MVC 컨트롤러로 인식된다.
- **@Repository** : 스프링 데이터 접근 계층으로 인식하고 해당 계층에서 발생하는 예외는 모두 **DataAccessException**으로 변환한다.
- **@Service** : 특별한 처리는 하지 않으나, 개발자들이 핵심 비즈니스 계층을 인식하는데 도움을 준다.
- **@Configuration** : 스프링 설정 정보로 인식하고 **bean** 등록, **singleton scope** 을 보장

@Bean vs @Component

- **@Bean** 개발자가 컨트롤이 불가능한 외부 라이브러리들을 **Bean**으로 등록하고 싶은 경우 에 사용된다.
 - 메소드 또는 어노테이션 단위에 붙일 수 있다.
- **@Component** 개발자가 직접 컨트롤이 가능한 클래스들의 경우에 사용된다.
 - 클래스 또는 인터페이스 단위에 붙일 수 있다.

Bean lifecycle



<https://media.geeksforgeeks.org/wp-content/uploads/20200428011831/Bean-Life-Cycle-Process-flow3.png>

생성자 안에서 무거운 초기화 작업을 함께 하는 것보다는 객체를 생성하는 부분과 초기화 하는 부분을 명확하게 나누는 것이 유지보수 관점에서 좋다.

* Bean

가

Bean lifecycle callback

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class ExampleBean {

    @PostConstruct
    public void initialize() throws Exception {
        // 초기화 콜백 (의존관계 주입이 끝나면 호출)
    }

    @PreDestroy : bean
    public void close() throws Exception {
        // 소멸 전 콜백 (메모리 반납, 연결 종료와 같은 과정)
    }
}
```

Bean scope

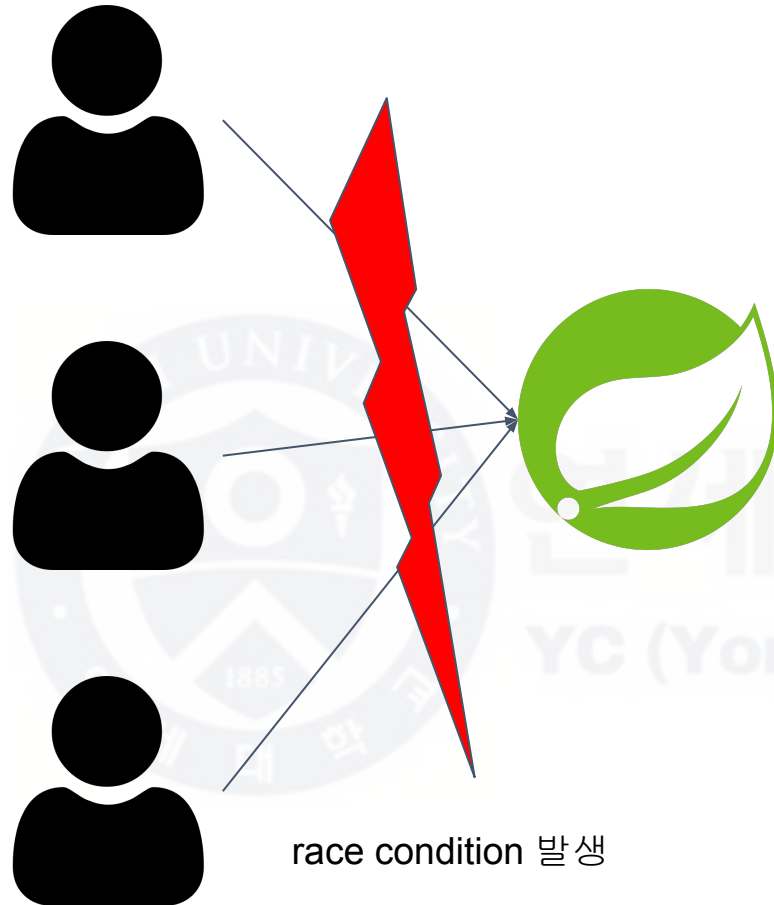
Singleton

- **spring** 컨테이너에서 한 번만 생성되며, 컨테이너가 사라질 때 제거된다.
- 생성된 하나의 인스턴스는 **Spring Beans Cache**에 저장되고, 해당 **bean**에 대한 요청과 참조가 있으며 캐시된 객체를 반환한다.
- 하나만 생성되기 때문에 동일 참조를 보장한다.
- 기본적으로 모든 **bean**은 스코프가 명시적으로 지정되지 않으면 싱글톤이다.
- 싱글톤 타입으로 적합한 객체
 - o 상태가 없는 공유
 - o 객체 읽기 전용으로만 상태를 가진 객체
 - o 쓰기가 가능한 상태를 지니면서도 사용 빈도가 매우 높은 객체 단, 이때는 동기화 전략이 필요함.

prototype

- **prototype bean**은 DI가 발생할 때마다 새로운 객체가 생성되어 주입된다.
- **bean** 소멸에 스프링 컨테이너가 관여하지 않고, **gc**에 의해 **bean**이 제거된다.

Bean scope



Spring bean 은 thread safe?

- spring은 요청 당 thread 생성하는 multi-threading 환경
- 요청마다 객체(bean)을 만들게 된다면 자원 낭비가 심함
- multi threading 에 따라 race condition 및 의도치 않은 동작 발생가능
- Spring 은 기본적으로 singleton bean 으로 만들어서 immutable 하므로 thread-safe 문제를 걱정할 필요가 일반적으로 없음 : bean 가 thread-safe

*

3

3

thread



thread-safe 하게 만드는 법

- stateless 하게 만듦
- synchronized 키워드 사용
- concurrent 객체 사용
- immutable 객체 사용

Bean annotation

DI 시 **bean** 이 여러 개일 경우

- 우선적으로, **bean** 의 **class** 를 기반으로 검색
- 같은 **class type** 이 여러개인 경우
 - o 오류 발생
 - o **@Autowired** 의 필드명 매칭
 - o **@Qualifier** : 명시적으로 **bean** 의 이름을 설정
 - o **@Primary** : 우선 순위 지정

예시

두 개의 데이터베이스 연결을 **bean** 으로 등록하는 경우

- 메인 데이터베이스 커백션을 사용하는 **bean** -> **@Primary**로 기본설정
- 서브 데이터베이스의 커백션을 사용하는 **bean** -> **@Qualifier** 지정해서 명시적으로 설정

YC (Yonsei X Codepresso) Tech Academy

Bean annotation

 이미 발생한 circular dependency 를 해결하려면?

- 재설계
 - 객체 지향에서 순환 참조는 테스트, 모듈화를 해치므로 매우 좋지 않은 형태
 - 적절한 기능 분배를 통해 모듈화 필요
- 어쩔 수 없다면?
 - `@Lazy`, `@PostConstruct` 등 주입 순서를 조정하는 방식으로 수정

`@Component`

```
public class CircularDependencyA {
```

```
    private CircularDependencyB circB;
```

```
    @Autowired
```

```
    public CircularDependencyA(@Lazy CircularDependencyB circB) {  
        this.circB = circB;
```

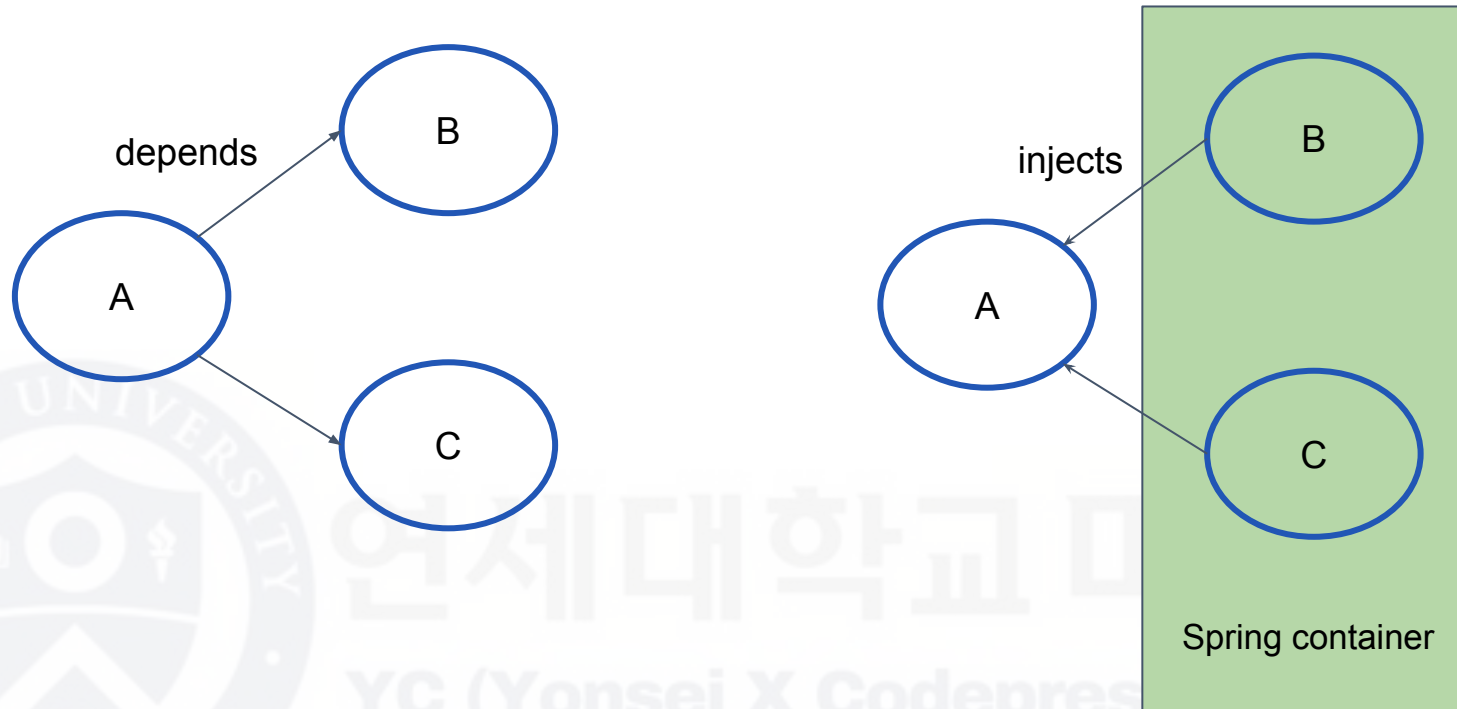
```
    }
```

```
}
```

Dependency Injection



Inversion of Control



- 프로그램의 더 큰 모듈성을 갖게 함
- 구성 요소를 격리시켜 프로그램을 더 쉽게 테스트하고, 의존성을 대체하거나 컴포넌트가 계약을 통해 통신할 수 있게 함.

Dependency Injection

일반적 코드

```
public class Store {  
    private Item item;  
  
    public Store() {  
        item = new ItemImpl1();  
    }  
}
```

Dependency Injection

```
public class Store {  
    private Item item;  
    public Store(Item item) {  
        this.item = item;  
    }  
}
```

DI(Dependency Injection)는 제어의 역전(IoC)을 구현하는 데 사용할 수 있는 패턴으로, 역전되는 제어는 객체의 의존성을 설정

Dependency Injection in Spring boot

Setter injection

```
public class SimpleMovieLister {  
  
    // the SimpleMovieLister has a dependency on the  
    // MovieFinder  
  
    private MovieFinder movieFinder;  
  
    // a setter method so that the Spring container can  
    // inject a MovieFinder  
  
    public void setMovieFinder(MovieFinder  
movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

- method의 parameter를 통해 의존성을 주입하는 방식
- Setter Injection은 선택적인 의존성을 사용할 때 유용하다.
상황에 따라 의존성 주입이 가능

Dependency Injection in Spring boot



Field injection

```
public class SimpleMovieLiser {  
  
    @Autowired  
    private MovieFinder movieFinder;  
}
```

- 의존성을 주입하기가 쉽다.
- 이때 많은 **field injection** 이 리팩토링을 해야한다는 신호가 될 수 있다.
 - 의존성 관계를 숨길 수 있어 명시적으로 어떤 **bean** 이 연관되어있는지 알기 어렵다.
 - 순환 참조(circular dependency)의 위험 존재

YC (Yonsei X Codepresso) Tech Academy

Dependency Injection in Spring boot

```
public class MyClass {  
    private final Book book;
```

```
    @Autowired
```

```
    public MyClass(Book book) {  
        this.book = book;
```

```
    }
```

```
}
```

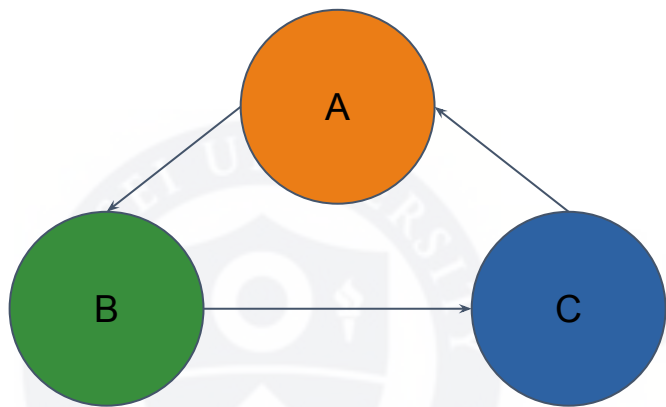
*



Constructor injection

- **Constructor Injection**은 필수적인 의존성 주입에 유용하다.
 - 단위 테스트 시 명시적으로 의존 관계를 표현하여 테스트 작성이 간편하다.
- **final**을 선언할 수 있으므로 객체가 불변하도록 할 수 있다.
 - **injection** 된 객체를 변경하는 것을 방지하여 안정성 확보
- 순환 의존성을 **compile** 단계에서 파악할 수 있다.

Circular dependency



증상

- 순환 참조 시 메소드 호출이 연속으로 발생하여 **stack** 이 넘치는 에러 (**stack overflow**) 발생
- 의도치 않은 동작 발생
- 느린 초기 시작 시간

* Spring

dependency

가


Spring



이미 발생한 circular dependency 를 해결하려면?

- 재설계
 - 객체 지향에서 순환 참조는 테스트, 모듈화를 해치므로 매우 좋지 않은 형태
 - 적절한 기능 분배를 통해 모듈화 필요
- 리팩토링할 시간이 없다면?
 - **@Lazy**, **@PostConstruct** 등 주입 순서를 조정하는 방식으로 수정

Circular dependency

 해결 예시

```
@Component
public class CircularDependencyA {

    private CircularDependencyB circB;

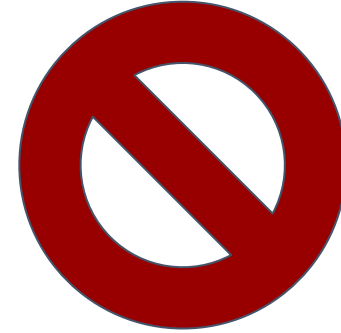
    @Autowired
    public CircularDependencyA(@Lazy CircularDependencyB circB) {
        this.circB = circB;
    }
}
```

연세대학교 미래교육원
YC (Yonsei X Codepresso) Tech Academy

DI 를 통한 간편한 테스트

DI 를 이용한 테스트

```
public class MemberService {  
  
    private final MemberRepository memberRepository = new  
    MemoryMemberRepository();  
  
}
```



```
public class MemberService {  
  
    private final MemberRepository memberRepository;  
  
    // 테스트 코드에서 같은 repository를 사용하기 위해 생성자를 사용  
    // (직접 생성하는 것이 아니라 외부에서 넣어주도록)  
    public MemberService(MemberRepository memberRepository) {  
        this.memberRepository = memberRepository;  
    }  
  
}
```

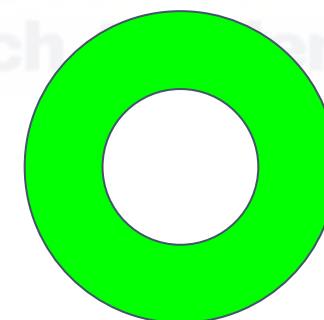


DI 를 이용한 테스트

```
class MemberServiceTest {  
  
    MemberService memberService = new MemberService();  
    MemoryMemberRepository memberRepository = new  
MemoryMemberRepository();  
}
```



```
class MemberServiceTest {  
  
    // main 코드와 test 코드에서 같은 repository를 사용하기 위함  
    @BeforeEach  
    public void beforeEach() {  
        memberRepository = new MemoryMemberRepository();  
        memberService = new MemberService(memberRepository);  
    }  
}
```



Test double

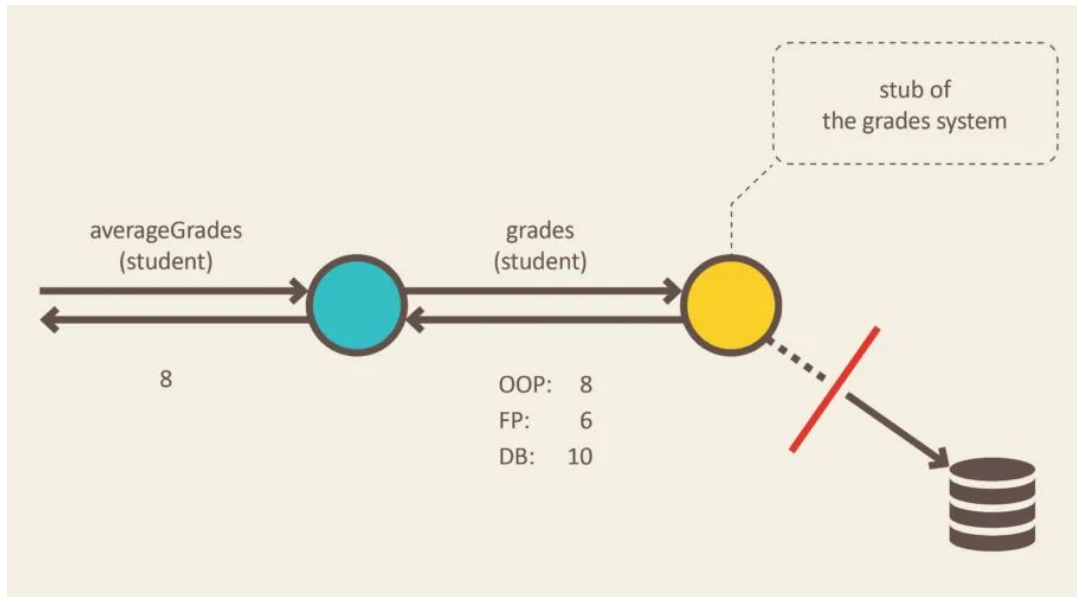


Test double



테스트를 진행하기 어려운 경우 이를 대신해 테스트를 진행할 수 있도록 만들어주는 객체
영화 촬영 시 위험한 역할을 대신하는 **Stunt double**에서 비롯되었다.

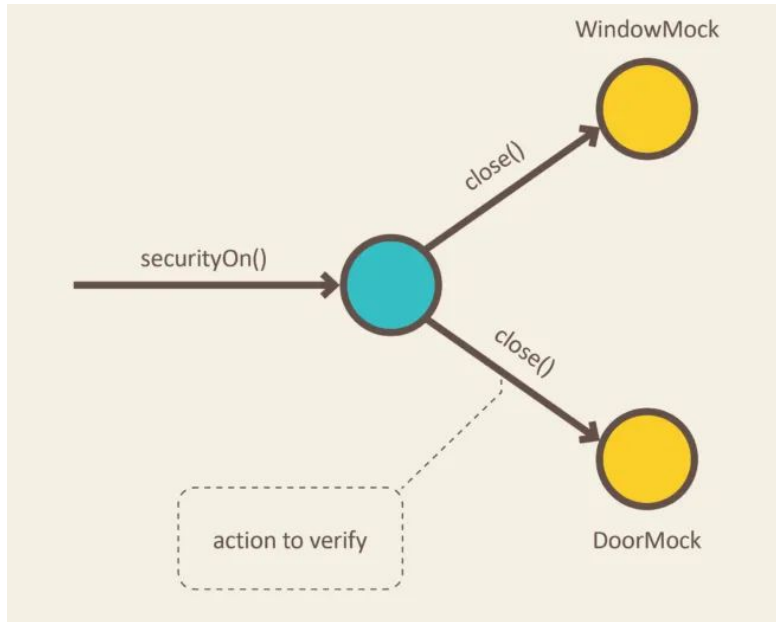
Test double



```
public class GradesServiceTest {  
    private Student studen = new Student();  
    private Gradebook gradebook = mock(Gradebook.class);  
  
    @Test  
    public void calculates_grades_average_for_student() {  
        when(gradebook.gradesFor(student)).thenReturn(grades(8, 6,  
10)); //stubbing gradebook  
        double averageGrades = new  
GradesService(gradebook).averageGrades(student);  
        assertEquals(8.0);  
    }  
}
```

- stub
 - 로직은 없고 단지 지정한 값을 반환하는 객체, 실제로 동작하는 것 처럼 동작한다.
 - **interface** 혹은 클래스를 최소한으로 구현된 상태, 호출된 요청에 대해 미리 준비해둔 결과를 리턴

Test double



```
public class SecurityCentralTest {
    Window windowMock = mock(Window.class);
    Door doorMock = mock(Door.class);

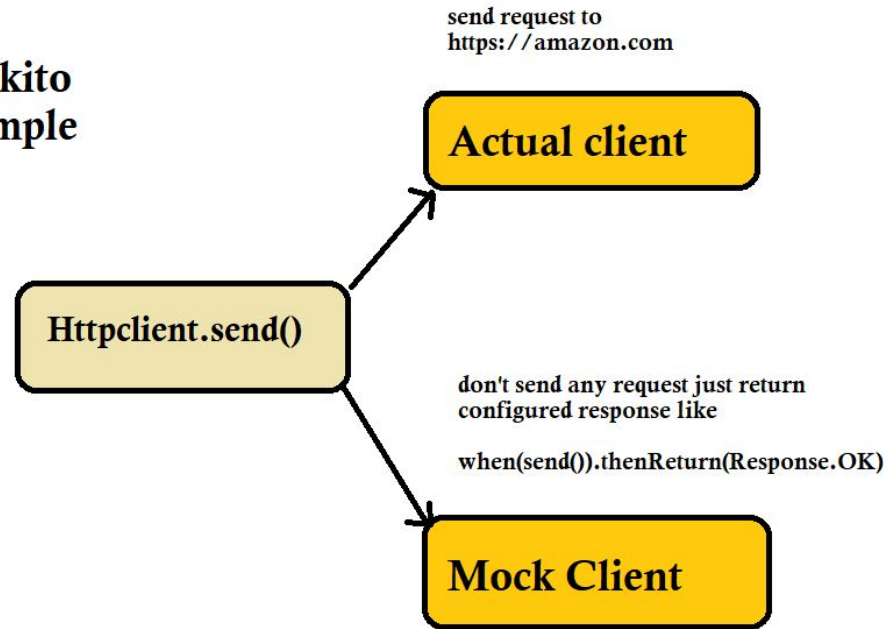
    @Test
    public void enabling_security_locks_windows_and_doors() {
        SecurityCentral securityCentral = new
        SecurityCentral(windowMock, doorMock);
        securityCentral.securityOn();
        verify(doorMock).close();
        verify(windowMock).close();
    }
}
```

- mock

- 테스트 하고자 하는 코드와 맞물려 동작하는 객체들을 대신하여 생성하는 껍데기만 존재하는 객체
- 테스트 대상 **mock** 객체는 내부의 가상화된 메소드를 호출할 수 있고, 메소드가 호출에 따라 어떤 결과를 리턴할지 결정할 수 있다.
- 객체 사이의 행위를 테스트하기 위해 사용

Mockito

Mockito Example Java



- **Mock** 객체를 쉽게 만들고, 관리하고, 검증할 수 있는 방법을 제공하는 프레임워크
 - **Mock**: 진짜 객체와 비슷하게 동작하지만, 프로그래머가 직접 행동을 관리하는 객체
- 애플리케이션에서 데이터베이스, 외부 **API** 등을 테스트할 때, 해당 제품들이 어떻게 작동하는지 항상 사용하면서 테스트를 작성한다면 매우 불편할 것이다. 이럴 때 어떻게 작동하는지 예측을 하여 **Mock** 객체를 만들어서 사용하면, 편리한 테스트가 가능하다.
- 이미 구현되어 있는 클래스는 **mock** 을 이용하여 테스트 작성할 필요가 없음. 예측하기 힘든 외부 서비스에 사용

추가적 테스트

@TestContainer

- 외부 의존성을 **docker** 로 가상화 된 이미지를 구동하여 테스트 환경 제공, **DB**, 외부 서비스 등 테스트 하기
힘든 환경을 **Mock** 하여 테스트
<https://java.testcontainers.org/>

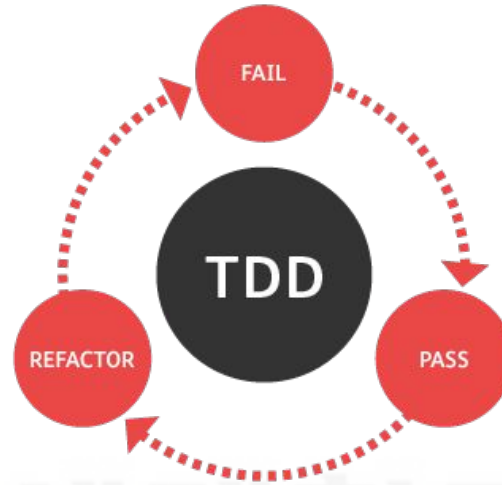
Selenium

- 웹 페이지 행동을 자동화 시키는 스크립트를 통해 테스트 환경 제공
<https://www.selenium.dev/>

Spring slice test



Spring slice test



* TDD: Test Driven Development

F.I.R.S.T

- **Fast** — 테스트는 빨라야 한다.
- **Isolated** — 각 테스트는 서로 의존하면 안된다.
- **Repeatable** — 테스트는 어떤 환경에서도 반복 가능해야 한다.
- **Self-validating** — 테스트는 **bool** 값으로 결과를 내야 한다.
- **Timely** — 테스트는 적시에 작성해야 한다.

Spring slice test



<https://i.stack.imgur.com/bmhZg.jpg>

단위 테스트

- 클래스 범위 내에서 작은 단위(메소드)의 기능에 대한 유효성 검증
 - 간단하고 명료하고 빠르게 실행되어야 함
 - 하나의 메소드에 하나 이상의 테스트가 존재 할 수 있음
 - **coverage** 가 높을 수록 신뢰도가 높음
 - 작게 나뉜 테스트는 해당 로직이 어떤 역할을 하는지 파악 가능하게 함

Spring slice test

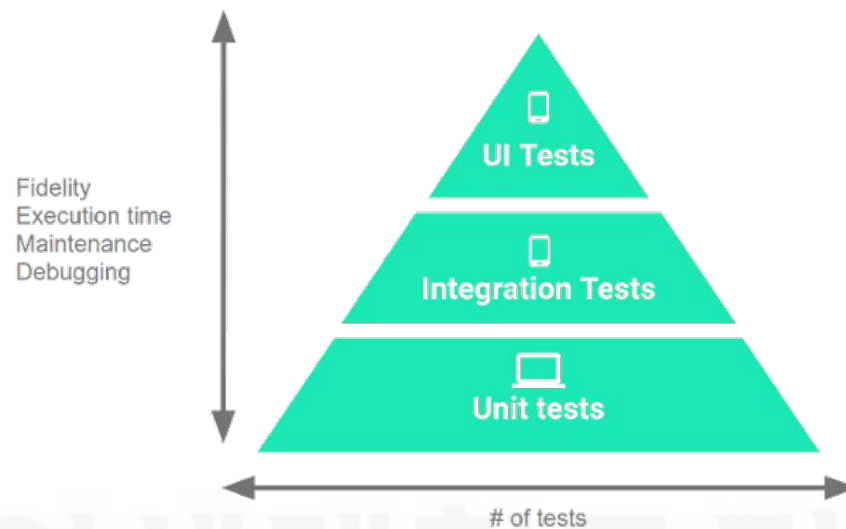


<https://i.stack.imgur.com/yHGn1.gif>

통합 테스트

- 서로 다른 모듈 혹은 클래스 간 상호작용의 유효성을 검사하는 테스트
- 이러한 통합 테스트가 필요한 이유는 각각 단위 테스트가 검증되었다 하더라도, 모듈 간 인터페이스 및 데이터 흐름이 의도한 대로 작동하지 않는 경우도 있기 때문이다. 그래서 조금 더 넓은 범위에서 추가적인 테스트가 필요
- 단위 테스트보다 테스트 코드를 작성하기가 복잡하다.

Spring slice test



<https://developer.android.com/static/images/training/testing/pyramid.png?hl=ko>

UI 테스트

- 대형 테스트
- 실제 사용자들이 사용하는 화면에 대한 테스트를 하여 서비스의 기능이 정상적으로 작동하는지 검증하는 테스트
- 실제 앱을 사용하는 사용자의 흐름에 대해 테스트 함으로써 **UI** 변경 사항으로 발생할 수 있는 문제를 사전에 차단

Spring slice test



Test slicing is about segmenting the `ApplicationContext` that is created for your test. Typically, if you want to test a controller using `MockMvc`, surely you don't want to bother with the data layer. Instead you'd probably want to mock the service that your controller uses and validate that all the web-related interaction works as expected.

To DO

- ❑ 요구사항 추출
- ❑ profile 별로 설정 나누기 실습
- ❑ controller 작성 & swagger 작성
- ❑ service 로직 구현
- ❑ 테스트 코드 작성
- ❑ Entity 설계

The background is a solid blue color with several abstract geometric elements. In the top-left and bottom-right corners, there are circular halftone patterns of varying densities. Diagonal lines in a lighter shade of blue cross these patterns. On the right side, there are two thin, light-blue circles. The overall design is modern and minimalist.

EOD