# Spring

## ApplicationContext

```
로컬 개발환경
테스트 개발환경
운영(Prod) 개발환경
Stage 개발환경
환경변수(로컬, 개발, 운영등 구분해서 처리 가능)
애플리케이션 이벤트
편리한 리로스 조회
```

## 자동 빈 등록과 수동 빈 등록 충돌시

개발자가 의도적으로 이런 결과를 기대했다면, 자동 보다는 수동이 우선권을 가지는 것이 좋다. 하지만 현실은 개발자가 의도적으로 설정해서 이런 결과가 만들어지기 보다는 여러 설정들이 꼬여서 이런 결과가 만들어지는 경우가 대부분이다!
그러면 정말 잡기 어려운 버그가 만들어진다. 항상 잡기 어려운 버그는 애매한 버그다.
그래서 최근 스프링 부트에서는 수동 빈 등록과 자동 빈 등록이 충돌나면 오류가 발생하도록 기본 값을 바꾸었다.

## 의존관계 주입(DI) 방법

생성자 의존 주입

이름 그대로 생성자를 통해서 의존 관계를 주입 받는 방법이다.

특징
생성자 호출시점에 딱 1번만 호출되는 것이 보장된다.
불변, 필수 의존관계에 사용

Setter 의존 주입

가끔씩 사용하는 의존 주입. 런타임 환경에서 객체를 변경할때 가끔 사용한다.

필드 의존 주입

DI 프레임워크가 없으면 아무것도 할 수 없다.
사용하지 말자!
애플리케이션의 실제 코드와 관계 없는 테스트 코드
스프링 설정을 목적으로 하는 @Configuration 같은 곳에서만 특별한 용도로 사용

일반 메서드 주입
일반 메서드를 통해서 주입 받을 수 있다.
특징

한번에 여러 필드를 주입 받을 수 있다.
일반적으로 잘 사용하지 않는다.

## Autowired 옵션

주입할 스프링 빈이 없어도 동작해야 할 때가 있다.
그런데 @Autowired 만 사용하면 required 옵션의 기본값이 true 로 되어 있어서 자동 주입 대상이 없으면 오류가 발생한다.
자동 주입 대상을 옵션으로 처리하는 방법은 다음과 같다.
@Autowired(required=false) : 자동 주입할 대상이 없으면 수정자 메서드 자체가 호출 안됨
org.springframework.lang.@Nullable : 자동 주입할 대상이 없으면 null이 입력된다.
Optional<> : 자동 주입할 대상이 없으면 Optional.empty 가 입력된다.

## 생성자 주입을 선택하라!

대부분의 의존관계 주입은 한번 일어나면 애플리케이션 종료시점까지 의존관계를 변경할 일이 없다. 오히려 대부분의 의존관계는 애플리케이션 종료 전까지 변하면 안된다.(불변해야 한다.)
수정자 주입을 사용하면, setXxx 메서드를 public으로 열어두어야 한다.
누군가 실수로 변경할 수 도 있고, 변경하면 안되는 메서드를 열어두는 것은 좋은 설계 방법이 아니다.
생성자 주입은 객체를 생성할 때 딱 1번만 호출되므로 이후에 호출되는 일이 없다. 따라서 불변하게 설계할 수 있다.

기억하자! 컴파일 오류는 세상에서 가장 빠르고, 좋은 오류다!

> 참고: 수정자 주입을 포함한 나머지 주입 방식은 모두 생성자 이후에 호출되므로, 필드에 final 키워드를 사용할 수 없다. 오직 생성자 주입 방식만 final 키워드를 사용할 수 있다.

정리
생성자 주입 방식을 선택하는 이유는 여러가지가 있지만, 프레임워크에 의존하지 않고, 순수한 자바 언어의 특징을 잘 살리는 방법이기도 하다.
기본으로 생성자 주입을 사용하고, 필수 값이 아닌 경우에는 수정자 주입 방식을 옵션으로 부여하면 된다.
생성자 주입과 수정자 주입을 동시에 사용할 수 있다.
항상 생성자 주입을 선택해라! 그리고 가끔 옵션이 필요하면 수정자 주입을 선택해라. 필드 주입은 사용하지 않는게 좋다.

## +@Lombok

@Data만 있는게 아니였다.

롬복 라이브러리가 제공하는 @RequiredArgsConstructor 기능을 사용하면 final이 붙은 필드를 모아서 생성자를 자동으로 만들어준다. (다음 코드에는 보이지 않지만 실제 호출 가능하다.)

최근에는 생성자를 딱 1개 두고, @Autowired 를 생략하는 방법을 주로 사용한다. 여기에 Lombok 라이브러리의 @RequiredArgsConstructor 함께 사용하면 기능은 다 제공하면서, 코드는 깔끔하게 사용할 수 있다.

```
plugins {
id 'org.springframework.boot' version '2.3.2.RELEASE'
id 'io.spring.dependency-management' version '1.0.9.RELEASE'
id 'java'
}
```

```
group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'
//lombok 설정 추가 시작
configurations {
compileOnly {
extendsFrom annotationProcessor
}
}
//lombok 설정 추가 끝
repositories {
mavenCentral()
}
dependencies {
implementation 'org.springframework.boot:spring-boot-starter'
//lombok 라이브러리 추가 시작
compileOnly 'org.projectlombok:lombok'
annotationProcessor 'org.projectlombok:lombok'
testCompileOnly 'org.projectlombok:lombok'
testAnnotationProcessor 'org.projectlombok:lombok'
//lombok 라이브러리 추가 끝
testImplementation('org.springframework.boot:spring-boot-starter-test') {
exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
}
```

1. Preferences(윈도우 File Settings) plugin lombok 검색 설치 실행 (재시작)
2. Preferences Annotation Processors 검색 Enable annotation processing 체크 (재시작)
3. 임의의 테스트 클래스를 만들고 @Getter, @Setter 확인

## Autowire 충돌시 @Qualifier, @Primary

조회 대상 빈이 2개 이상일 때 해결 방법
@Autowired 필드 명 매칭
@Qualifier @Qualifier끼리 매칭 빈 이름 매칭
@Primary 사용

@Autowired 매칭 정리

1. 타입 매칭
2. 타입 매칭의 결과가 2개 이상일 때 필드 명, 파라미터 명으로 빈 이름 매칭

@Qualifier 정리

1. @Qualifier끼리 매칭
2. 빈 이름 매칭
3. NoSuchBeanDefinitionException 예외 발생

@Primary, @Qualifier 활용
코드에서 자주 사용하는 메인 데이터베이스의 커넥션을 획득하는 스프링 빈이 있고, 코드에서 특별한 기능으로 가끔 사용하는 서브 데이터베이스의 커넥션을 획득하는 스프링 빈이 있다고 생각해보자. 메인 데이터베이스의 커넥션을 획득하는 스프링 빈은 @Primary 를 적용해서 조회하는 곳에서 @Qualifier 지정 없이 편리하게 조회하고, 서브 데이터베이스 커넥션 빈을 획득할 때는 @Qualifier 를 지정해서 명시적으로 획득 하는 방식으로 사용하면 코드를 깔끔하게 유지할 수 있다. 물론 이때 메인 데이터베이스의 스프링 빈을 등록할 때 @Qualifier 를 지정해주는 것은 상관없다.

우선순위
@Primary 는 기본값 처럼 동작하는 것이고, @Qualifier 는 매우 상세하게 동작한다. 이런 경우 어떤 것이 우선권을 가져갈까? 스프링은 자동보다는 수동이, 넓은 범위의 선택권 보다는 좁은 범위의 선택권이 우선순위가 높다. 따라서 여기서도 @Qualifier 가 우선권이 높다.

- 

애노테이션에는 상속이라는 개념이 없다. 이렇게 여러 애노테이션을 모아서 사용하는 기능은 스프링이 지원해주는 기능이다. @Qulifier 뿐만 아니라 다른 애노테이션들도 함께 조합해서 사용할 수 있다.
단적으로 @Autowired도 재정의 할 수 있다. 물론 스프링이 제공하는 기능을 뚜렷한 목적 없이 무분별하게 재정의 하는 것은 유지보수에 더 혼란만 가중할 수 있다.

## 조회한 빈이 모두 필요할때 → List, Map

Autowired 주입시 빈을 모두 사용해야 할 땐

Map<String, InterfaceName>으로 받아서 사용  가능하다.

```
package hello.core.autowired;

import hello.core.AutoAppConfig;
import hello.core.discount.DiscountPolicy;
import hello.core.member.Grade;
import hello.core.member.Member;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.test.context.event.annotation.AfterTestClass;

import java.util.List;
import java.util.Map;

import static org.assertj.core.api.Assertions.assertThat;

public class AllBeanTest {



    @Test
    void findAllBean(){
        ApplicationContext ac = new AnnotationConfigApplicationContext(AutoAppConfig.class, DiscountService.class);

        DiscountService discountService = ac.getBean(DiscountService.class);
        Member member = new Member(1L, "userA", Grade.VIP);
        int discountPrice = discountService.discount(member, 10000, "fixDiscountPolicy");

        assertThat(discountService).isInstanceOf(DiscountService.class);
        assertThat(discountPrice).isEqualTo(1000);

    }

    static class DiscountService{
        private final Map<String, DiscountPolicy> policyMap;
        private final List<DiscountPolicy> policies;

        @Autowired
        public DiscountService(Map<String, DiscountPolicy> policyMap, List<DiscountPolicy> policies) {
            this.policyMap = policyMap;
            this.policies = policies;
            System.out.println("policyMap = " + policyMap);
            System.out.println("policies = " + policies);
        }

        public int discount(Member member, int price, String discountCode) {
            DiscountPolicy discountPolicy = policyMap.get(discountCode);
            return discountPolicy.discount(member, price);
        }
```

```
    }
}
```

로직 분석
DiscountService는 Map으로 모든 DiscountPolicy 를 주입받는다. 이때 fixDiscountPolicy , rateDiscountPolicy 가 주입된다.
discount () 메서드는 discountCode로 "fixDiscountPolicy"가 넘어오면 map에서 fixDiscountPolicy 스프링 빈을 찾아서 실행한다. 물론 “rateDiscountPolicy”가 넘어오면 rateDiscountPolicy 스프링 빈을 찾아서 실행한다.
주입 분석
Map<String, DiscountPolicy> : map의 키에 스프링 빈의 이름을 넣어주고, 그 값으로 DiscountPolicy 타입으로 조회한 모든 스프링 빈을 담아준다.
List<DiscountPolicy> : DiscountPolicy 타입으로 조회한 모든 스프링 빈을 담아준다.
만약 해당하는 타입의 스프링 빈이 없으면, 빈 컬렉션이나 Map을 주입한다.

## 자동, 수동 등록 기준

편리한 자동 기능을 기본으로 사용하자
그러면 어떤 경우에 컴포넌트 스캔과 자동 주입을 사용하고, 어떤 경우에 설정 정보를 통해서 수동으로 빈을 등록하고, 의존관계도 수동으로 주입해야 할까?
결론부터 이야기하면, 스프링이 나오고 시간이 갈 수록 점점 자동을 선호하는 추세다. 스프링은 @Component 뿐만 아니라 @Controller , @Service , @Repository 처럼 계층에 맞추어 일반적인 애플리케이션 로직을 자동으로 스캔할 수 있도록 지원한다. 거기에 더해서 최근 스프링 부트는 컴포넌트 스캔을 기본으로 사용하고, 스프링 부트의 다양한 스프링 빈들도 조건이 맞으면 자동으로 등록하도록 설계했다.
설정 정보를 기반으로 애플리케이션을 구성하는 부분과 실제 동작하는 부분을 명확하게 나누는 것이 이상적이지만, 개발자 입장에서 스프링 빈을 하나 등록할 때 @Component 만 넣어주면 끝나는 일을 @Configuration 설정 정보에 가서 @Bean 을 적고, 객체를 생성하고, 주입할 대상을 일일이 적어주는 과정은 상당히 번거롭다.
또 관리할 빈이 많아서 설정 정보가 커지면 설정 정보를 관리하는 것 자체가 부담이 된다.
그리고 결정적으로 자동 빈 등록을 사용해도 OCP, DIP를 지킬 수 있다.
그러면 수동 빈 등록은 언제 사용하면 좋을까?
애플리케이션은 크게 업무 로직과 기술 지원 로직으로 나눌 수 있다.

업무 로직 빈: 웹을 지원하는 컨트롤러, 핵심 비즈니스 로직이 있는 서비스, 데이터 계층의 로직을 처리하는 리포지토리 등이 모두 업무 로직이다. 보통 비즈니스 요구사항을 개발할 때 추가되거나 변경된다.
기술 지원 빈: 기술적인 문제나 공통 관심사(AOP)를 처리할 때 주로 사용된다. 데이터베이스 연결이나, 공통 로그 처리처럼 업무 로직을 지원하기 위한 하부 기술이나 공통 기술들이다.

업무 로직은 숫자도 매우 많고, 한번 개발해야 하면 컨트롤러, 서비스, 리포지토리 처럼 어느정도 유사한 패턴이 있다. 이런 경우 자동 기능을 적극 사용하는 것이 좋다. 보통 문제가 발생해도 어떤 곳에서 문제가 발생했는지 명확하게 파악하기 쉽다.
기술 지원 로직은 업무 로직과 비교해서 그 수가 매우 적고, 보통 애플리케이션 전반에 걸쳐서 광범위하게 영향을 미친다. 그리고 업무 로직은 문제가 발생했을 때 어디가 문제인지 명확하게 잘 들어나지만, 기술 지원 로직은 적용이 잘 되고 있는지 아닌지 조차 파악하기 어려운 경우가 많다. 그래서 이런 기술 지원 로직들은 가급적 수동 빈 등록을 사용해서 명확하게 들어내는 것이 좋다.
애플리케이션에 광범위하게 영향을 미치는 기술 지원 객체는 수동 빈으로 등록해서 딱! 설정 정보에 바로 나타나게 하는 것이 유지보수 하기 좋다.
비즈니스 로직 중에서 다형성을 적극 활용할 때 의존관계 자동 주입 - 조회한 빈이 모두 필요할 때, List, Map을 다시 보자.
DiscountService 가 의존관계 자동 주입으로 Map<String, DiscountPolicy> 에 주입을 받는 상황을

생각해보자. 여기에 어떤 빈들이 주입될 지, 각 빈들의 이름은 무엇일지 코드만 보고 한번에 쉽게 파악할 수 있을까? 내가 개발했으니 크게 관계가 없지만, 만약 이 코드를 다른 개발자가 개발해서 나에게 준 것이라면 어떨까?
자동 등록을 사용하고 있기 때문에 파악하려면 여러 코드를 찾아봐야 한다.
이런 경우 수동 빈으로 등록하거나 또는 자동으로하면 특정 패키지에 같이 묶어두는게 좋다! 핵심은 딱 보고 이해가 되어야 한다!

참고로 스프링과 스프링 부트가 자동으로 등록하는 수 많은 빈들은 예외다. 이런 부분들은 스프링 자체를 잘 이해하고 스프링의 의도대로 잘 사용하는게 중요하다. 스프링 부트의 경우 DataSource 같은 데이터베이스 연결에 사용하는 기술 지원 로직까지 내부에서 자동으로 등록하는데, 이런 부분은 메뉴얼을 잘 참고해서 스프링 부트가 의도한 대로 편리하게 사용하면 된다. 반면에 스프링 부트가 아니라 내가 직접 기술 지원 객체를 스프링 빈으로 등록한다면 수동으로 등록해서 명확하게 들어내는 것이 좋다.

정리
편리한 자동 기능을 기본으로 사용하자
직접 등록하는 기술 지원 객체는 수동 등록
다형성을 적극 활용하는 비즈니스 로직은 수동 등록을 고민해보자

## 빈 라이프사이클

스프링 빈은 간단하게 다음과 같은 라이프사이클을 가진다.
객체 생성 의존관계 주입
스프링 빈은 객체를 생성하고, 의존관계 주입이 다 끝난 다음에야 필요한 데이터를 사용할 수 있는 준비가 완료된다. 따라서 초기화 작업은 의존관계 주입이 모두 완료되고 난 다음에 호출해야 한다. 그런데 개발자가 의존관계 주입이 모두 완료된 시점을 어떻게 알 수 있을까?
스프링은 의존관계 주입이 완료되면 스프링 빈에게 콜백 메서드를 통해서 초기화 시점을 알려주는 다양한 기능을 제공한다. 또한 스프링은 스프링 컨테이너가 종료되기 직전에 소멸 콜백을 준다. 따라서 안전하게 종료 작업을 진행할 수 있다.
스프링 빈의 이벤트 라이프사이클
스프링 컨테이너 생성 스프링 빈 생성 의존관계 주입 초기화 콜백 사용 소멸전 콜백 스프링 종료
초기화 콜백: 빈이 생성되고, 빈의 의존관계 주입이 완료된 후 호출
소멸전 콜백: 빈이 소멸되기 직전에 호출
스프링은 다양한 방식으로 생명주기 콜백을 지원한다.

> 참고: 객체의 생성과 초기화를 분리하자.
> 생성자는 필수 정보(파라미터)를 받고, 메모리를 할당해서 객체를 생성하는 책임을 가진다. 반면에 초기화는
> 이렇게 생성된 값들을 활용해서 외부 커넥션을 연결하는등 무거운 동작을 수행한다.
> 따라서 생성자 안에서 무거운 초기화 작업을 함께 하는 것 보다는 객체를 생성하는 부분과 초기화 하는
> 부분을 명확하게 나누는 것이 유지보수 관점에서 좋다. 물론 초기화 작업이 내부 값들만 약간 변경하는
> 정도로 단순한 경우에는 생성자에서 한번에 다 처리하는게 더 나을 수 있다.
> 참고: 싱글톤 빈들은 스프링 컨테이너가 종료될 때 싱글톤 빈들도 함께 종료되기 때문에 스프링 컨테이너가
> 종료되기 직전에 소멸전 콜백이 일어난다. 뒤에서 설명하겠지만 싱글톤 처럼 컨테이너의 시

> 작과 종료까지
> 생존하는 빈도 있지만, 생명주기가 짧은 빈들도 있는데 이 빈들은 컨테이너와 무관하게 해당 빈이 종료되기
> 직전에 소멸전 콜백이 일어난다. 자세한 내용은 스코프에서 알아보겠다.

스프링은 크게 3가지 방법으로 빈 생명주기 콜백을 지원한다.
인터페이스(InitializingBean, DisposableBean)
설정 정보에 초기화 메서드, 종료 메서드 지정
@PostConstruct, @PreDestroy 애노테이션 지원

## 빈 생명주기 콜백

인터페이스 InitializingBean, DisposableBean

InitializingBean 은 afterPropertiesSet() 메서드로 초기화를 지원한다.
DisposableBean 은 destroy() 메서드로 소멸을 지원한다.

초기화, 소멸 인터페이스 단점
이 인터페이스는 스프링 전용 인터페이스다. 해당 코드가 스프링 전용 인터페이스에 의존한다.

초기화, 소멸 메서드의 이름을 변경할 수 없다.
내가 코드를 고칠 수 없는 외부 라이브러리에 적용할 수 없다.

> 참고: 인터페이스를 사용하는 초기화, 종료 방법은 스프링 초창기에 나온 방법들이고, 지금은 다음의 더 나은 방법들이 있어서 거의 사용하지 않는다.

빈 등록 초기화, 소멸 메서드 지정
설정 정보에 @Bean(initMethod = "init", destroyMethod = "close") 처럼 초기화, 소멸 메서드를 지정할 수 있다.

설정 정보 사용 특징
메서드 이름을 자유롭게 줄 수 있다.
스프링 빈이 스프링 코드에 의존하지 않는다.
코드가 아니라 설정 정보를 사용하기 때문에 코드를 고칠 수 없는 외부 라이브러리에도 초기화, 종료 메서드를 적용할 수 있다.
종료 메서드 추론
@Bean의 destroyMethod 속성에는 아주 특별한 기능이 있다.
라이브러리는 대부분 close , shutdown 이라는 이름의 종료 메서드를 사용한다.
@Bean의 destroyMethod 는 기본값이 (inferred) (추론)으로 등록되어 있다.
이 추론 기능은 close , shutdown 라는 이름의 메서드를 자동으로 호출해준다. 이름 그대로 종료 메서드를 추론해서 호출해준다.
따라서 직접 스프링 빈으로 등록하면 종료 메서드는 따로 적어주지 않아도 잘 동작한다.
추론 기능을 사용하기 싫으면 destroyMethod="" 처럼 빈 공백을 지정하면 된다.

애노테이션 @PostConstruct, @PreDestroy

@PostConstruct , @PreDestroy 이 두 애노테이션을 사용하면 가장 편리하게 초기화와 종료를 실행할 수 있다.
@PostConstruct, @PreDestroy 애노테이션 특징

최신 스프링에서 가장 권장하는 방법이다.
애노테이션 하나만 붙이면 되므로 매우 편리하다.
패키지를 잘 보면 javax.annotation.PostConstruct 이다. 스프링에 종속적인 기술이 아니라 JSR-250 라는 자바 표준이다. 따라서 스프링이 아닌 다른 컨테이너에서도 동작한다.
컴포넌트 스캔과 잘 어울린다.
유일한 단점은 외부 라이브러리에는 적용하지 못한다는 것이다. 외부 라이브러리를 초기화, 종료 해야 하면 @Bean의 기능을 사용하자.

정리
@PostConstruct, @PreDestroy 애노테이션을 사용하자
코드를 고칠 수 없는 외부 라이브러리를 초기화, 종료해야 하면 @Bean 의 initMethod , destroyMethod 를 사용하자.

## 빈 스코프

스프링은 다음과 같은 다양한 스코프를 지원한다.
싱글톤: 기본 스코프, 스프링 컨테이너의 시작과 종료까지 유지되는 가장 넓은 범위의 스코프이다.
프로토타입: 스프링 컨테이너는 프로토타입 빈의 생성과 의존관계 주입까지만 관여하고 더는 관리하지 않는 매우 짧은 범위의 스코프이다.

웹 관련 스코프
request: 웹 요청이 들어오고 나갈때 까지 유지되는 스코프이다.

session: 웹 세션이 생성되고 종료될 때 까지 유지되는 스코프이다.
application: 웹의 서블릿 컨텍스트와 같은 범위로 유지되는 스코프이다.

프로토타입 스코프
싱글톤 스코프의 빈을 조회하면 스프링 컨테이너는 항상 같은 인스턴스의 스프링 빈을 반환한다. 반면에 프로토타입 스코프를 스프링 컨테이너에 조회하면 스프링 컨테이너는 항상 새로운 인스턴스를 생성해서 반환한다.

싱글톤 빈 요청

1. 싱글톤 스코프의 빈을 스프링 컨테이너에 요청한다.
2. 스프링 컨테이너는 본인이 관리하는 스프링 빈을 반환한다.
3. 이후에 스프링 컨테이너에 같은 요청이 와도 같은 객체 인스턴스의 스프링 빈을 반환한다.

프로토타입 빈 요청1

1. 프로토타입 스코프의 빈을 스프링 컨테이너에 요청한다.
2. 스프링 컨테이너는 이 시점에 프로토타입 빈을 생성하고, 필요한 의존관계를 주입한다.
3. 스프링 컨테이너는 생성한 프로토타입 빈을 클라이언트에 반환한다.
4. 이후에 스프링 컨테이너에 같은 요청이 오면 항상 새로운 프로토타입 빈을 생성해서 반환한다.

정리
여기서 핵심은 스프링 컨테이너는 프로토타입 빈을 생성하고, 의존관계 주입, 초기화까지만 처리한다는 것이다. 클라이언트에 빈을 반환하고, 이후 스프링 컨테이너는 생성된 프로토타입 빈을 관리하지 않는다.
프로토타입 빈을 관리할 책임은 프로토타입 빈을 받은 클라이언트에 있다. 그래서 @PreDestroy 같은 종료 메서드가 호출되지 않는다.

싱글톤 빈은 스프링 컨테이너 생성 시점에 초기화 메서드가 실행 되지만, 프로토타입 스코프의 빈은 스프링 컨테이너에서 빈을 조회할 때 생성되고, 초기화 메서드도 실행된다.
프로토타입 빈을 2번 조회했으므로 완전히 다른 스프링 빈이 생성되고, 초기화도 2번 실행된 것을 확인할 수 있다.
싱글톤 빈은 스프링 컨테이너가 관리하기 때문에 스프링 컨테이너가 종료될 때 빈의 종료 메서드가
실행되지만, 프로토타입 빈은 스프링 컨테이너가 생성과 의존관계 주입 그리고 초기화 까지만 관여하고, 더는 관리하지 않는다. 따라서 프로토타입 빈은 스프링 컨테이너가 종료될 때 @PreDestroy 같은 종료 메서드가 전혀 실행되지 않는다.

프로토타입 빈의 특징 정리
스프링 컨테이너에 요청할 때 마다 새로 생성된다.
스프링 컨테이너는 프로토타입 빈의 생성과 의존관계 주입 그리고 초기화까지만 관여한다.
종료 메서드가 호출되지 않는다.
그래서 프로토타입 빈은 프로토타입 빈을 조회한 클라이언트가 관리해야 한다. 종료 메서드에 대한 호출도 클라이언트가 직접 해야한다.

프로토타입 스코프 - 싱글톤 빈과 함께 사용시 문제점

스프링 컨테이너에 프토토타입 스코프의 빈을 요청하면 항상 새로운 객체 인스턴스를 생성해서 반환한다.
하지만 싱글톤 빈과 함께 사용할 때는 의도한 대로 잘 동작하지 않으므로 주의해야 한다

1. 클라이언트A는 스프링 컨테이너에 프로토타입 빈을 요청한다.
2. 스프링 컨테이너는 프로토타입 빈을 새로 생성해서 반환(x01)한다. 해당 빈의 count 필드 값은 0이다.
3. 클라이언트는 조회한 프로토타입 빈에 addCount() 를 호출하면서 count 필드를 +1 한다.
   결과적으로 프로토타입 빈(x01)의 count는 1이 된다.
   스프링 컨테이너에 프로토타입 빈 직접 요청2
4. 클라이언트B는 스프링 컨테이너에 프로토타입 빈을 요청한다.
5. 스프링 컨테이너는 프로토타입 빈을 새로 생성해서 반환(x02)한다. 해당 빈의 count 필드 값은 0이다.
6. 클라이언트는 조회한 프로토타입 빈에 addCount() 를 호출하면서 count 필드를 +1 한다.
   결과적으로 프로토타입 빈(x02)의 count는 1이 된다.

싱글톤 빈에서 프로토타입 빈 사용
이번에는 clientBean 이라는 싱글톤 빈이 의존관계 주입을 통해서 프로토타입 빈을 주입받아서 사용하는 예를 보자 .

clientBean 은 싱글톤이므로, 보통 스프링 컨테이너 생성 시점에 함께 생성되고, 의존관계 주입도
발생한다.

1. clientBean 은 의존관계 자동 주입을 사용한다. 주입 시점에 스프링 컨테이너에 프로토타입 빈을
   요청한다.
2. 스프링 컨테이너는 프로토타입 빈을 생성해서 clientBean 에 반환한다. 프로토타입 빈의 count 필드 값은 0이다.
   이제 clientBean 은 프로토타입 빈을 내부 필드에 보관한다. (정확히는 참조값을 보관한다.)

   클라이언트 A는 clientBean 을 스프링 컨테이너에 요청해서 받는다.싱글톤이므로 항상 같은
   clientBean 이 반환된다.
3. 클라이언트 A는 clientBean.logic() 을 호출한다.
4. clientBean 은 prototypeBean 의 addCount() 를 호출해서 프로토타입 빈의 count를 증가한다.
   count값이 1이 된다.

clientBean 이 반환된다.
클라이언트 B는 clientBean 을 스프링 컨테이너에 요청해서 받는다.싱글톤이므로 항상 같은
여기서 중요한 점이 있는데, clientBean 이 내부에 가지고 있는 프로토타입 빈은 이미 과거에 주입이 끝난 빈이다. 주입 시점에 스프링 컨테이너에 요청해서 프로토타입 빈이 새로 생성이 된 것이지, 사용 할 때마다 새로 생성되는 것이 아니다!
5. 클라이언트 B는 clientBean.logic() 을 호출한다.
6. clientBean 은 prototypeBean 의 addCount() 를 호출해서 프로토타입 빈의 count를 증가한다.
원래 count 값이 1이었으므로 2가 된다.

스프링은 일반적으로 싱글톤 빈을 사용하므로 , 싱글톤 빈이 프로토타입 빈을 사용하게 된다 . 그런데 싱글톤 빈은 생성 시점에만 의존관계 주입을 받기 때문에 , 프로토타입 빈이 새로 생성되기는 하지만 , 싱글톤 빈과 함께 계속 유지되는 것이 문제다 .
아마 원하는 것이 이런 것은 아닐 것이다 . 프로토타입 빈을 주입 시점에만 새로 생성하는게 아니라 , 사용할 때 마다 새로 생성해서 사용하는 것을 원할 것이다 .

> 참고: 여러 빈에서 같은 프로토타입 빈을 주입 받으면 , 주입 받는 시점에 각각 새로운 프로토타입 빈이 생성된다. 예를 들어서 clientA, clientB 가 각각 의존관계 주입을 받으면 각각 다른 인스턴스의 프로토타입 빈을 주입 받는다 .
> clientA prototypeBean@x01 > clientB
> prototypeBean@x02 > 물론 사용할 때 마다 새로 생성되는 것은 아니다 .

싱글톤 빈과 프로토타입 빈을 함께 사용할 때 , 어떻게 하면 사용할 때 마다 항상 새로운 프로토타입 빈을 생성할 수 있을까 ?
가장 간단한 방법은 싱글톤 빈이 프로토타입을 사용할 때 마다 스프링 컨테이너에 새로 요청하는 것이다 .

ApplicationContext 필드에 @Autowired걸기..

실행해보면ac.getBean() 을통해서항상새로운프로토타입빈이생성되는것을확인할수있다. 의존관계를 외부에서 주입(DI) 받는게 아니라 이렇게 직접 필요한 의존관계를 찾는 것을 Dependency Lookup (DL) 의존관계 조회 (탐색) 이라 한다. 그런데 이렇게 스프링의 애플리케이션 컨텍스트 전체를 주입받게 되면 , 스프링 컨테이너에 종속적인 코드가 되고, 단위 테스트도 어려워진다 . 지금 필요한 기능은 지정한 프로토타입 빈을 컨테이너에서 대신 찾아주는 딱 ! DL 정도의 기능만 제공하는 무언가가 있으면 된다 .
스프링에는 이미 모든게 준비되어 있다 .
ObjectFactory, ObjectProvider
지정한빈을컨테이너에서대신찾아주는 DL 서비스를제공하는것이바로ObjectProvider 이다. 참고로과거에는 ObjectFactory 가있었는데, 여기에편의기능을추가해서ObjectProvider 가만들어졌다.

실행해보면prototypeBeanProvider.getObject() 을통해서항상새로운프로토타입빈이생성되는
것을 확인할 수 있다 .
ObjectProvider 의getObject() 를호출하면내부에서는스프링컨테이너를통해해당빈을찾아서
반환한다. (DL) 스프링이 제공하는 기능을 사용하지만 , 기능이 단순하므로 단위테스트를 만들거나 mock 코드를 만들기는 훨씬 쉬워진다 .
ObjectProvider 는지금딱필요한 DL 정도의기능만제공한다.

ObjectFactory: 기능이 단순 , 별도의 라이브러리 필요 없음 , 스프링에 의존 ObjectProvider: ObjectFactory 상속, 옵션, 스트림 처리등 편의 기능이 많고 , 별도의 라이브러리 필요 없음, 스프링에 의존

정리

그러면 프로토타입 빈을 언제 사용할까 ? 매번 사용할 때 마다 의존관계 주입이 완료된 새로운 객체가 필요하면 사용하면 된다 . 그런데 실무에서 웹 애플리케이션을 개발해보면 , 싱글톤 빈으로 대부분의 문제를 해결할 수 있기 때문에 프로토타입 빈을 직접적으로 사용하는 일은 매우 드물다 . ObjectProvider , JSR330 Provider 등은 프로토타입 뿐만 아니라 DL이 필요한 경우는 언제든지
사용할 수 있다 .

> 참고: 스프링이 제공하는 메서드에 @Lookup 애노테이션을 사용하는 방법도 있지만 , 이전 방법들로
> 충분하고, 고려해야할 내용도 많아서 생략하겠다 .
> 참고: 실무에서 자바 표준인 JSR-330 Provider 를 사용할 것인지 , 아니면 스프링이 제공하는 ObjectProvider 를 사용할 것인지 고민이 될 것이다 . ObjectProvider 는 DL을 위한 편의 기능을 많이 제공해주고 스프링 외에 별도의 의존관계 추가가 필요 없기 때문에 편리하다 . 만약(정말 그럴일은 거의 없겠지만) 코드를 스프링이 아닌 다른 컨테이너에서도 사용할 수 있어야 한다면 JSR-330 Provider 를 사용해야한다.스프링을 사용하다 보면 이 기능 뿐만 아니라 다른 기능들도 자바 표준과 스프링이 제공하는 기능이 겹칠때가 많이 있다 . 대부분 스프링이 더 다양하고 편리한 기능을 제공해주기 때문에 , 특별히 다른 컨테이너를 사용할 일이 없다면 , 스프링이 제공하는 기능을 사용하면 된다 .

## 웹 스코프

웹 스코프의 특징
웹 스코프는 웹 환경에서만 동작한다.
웹 스코프는 프로토타입과 다르게 스프링이 해당 스코프의 종료시점까지 관리한다. 따라서 종료 메서드가 호출된다.
웹 스코프 종류
request: HTTP 요청 하나가 들어오고 나갈 때 까지 유지되는 스코프, 각각의 HTTP 요청마다 별도의 빈 인스턴스가 생성되고, 관리된다.
session: HTTP Session과 동일한 생명주기를 가지는 스코프
application: 서블릿 컨텍스트( ServletContext )와 동일한 생명주기를 가지는 스코프
websocket: 웹 소켓과 동일한 생명주기를 가지는 스코프
사실 세션이나, 서블릿 컨텍스트, 웹 소켓 같은 용어를 잘 모르는 분들도 있을 것이다. 여기서는 request 스코프를 예제로 설명하겠다. 나머지도 범위만 다르지 동작 방식은 비슷하다.

> 참고: 스프링 부트는 웹 라이브러리가 없으면 우리가 지금까지 학습한 AnnotationConfigApplicationContext 을 기반으로 애플리케이션을 구동한다 . 웹 라이브러리가 추가되면 웹과 관련된 추가 설정과 환경들이 필요하므로 AnnotationConfigServletWebServerApplicationContext 를기반으로애플리케이션을구동한다.

참고: requestURL 을 MyLogger 에 저장하는 부분은 컨트롤러 보다는 공통 처리가 가능한 스프링 인터셉터나 서블릿 필터 같은 곳을 활용하는 것이 좋다 . 여기서는 예제를 단순화하고 , 아직 스프링 인터셉터를 학습하지 않은 분들을 위해서 컨트롤러를 사용했다 . 스프링 웹에 익숙하다면 인터셉터를 사용해서 구현해보자 .

여기서 중요한점이 있다 . request scope 를 사용하지 않고 파라미터로 이 모든 정보를 서비스 계층에 넘긴다면, 파라미터가 많아서 지저분해진다 . 더 문제는 requestURL 같은 웹과 관련된 정보가 웹과 관련없는 서비스 계층까지 넘어가게 된다 . 웹과 관련된 부분은 컨트롤러까지만 사용해야 한다 . 서비스 계층은 웹 기술에 종속되지 않고 , 가급적 순수하게 유지하는 것이 유지보수 관점에서 좋다 . request scope 의 MyLogger 덕분에 이런 부분을 파라미터로 넘기지 않고 , MyLogger 의 멤버변수에 저장해서 코드와 계층을 깔끔하게 유지할 수 있다 .

스프링 애플리케이션을 실행하는 시점에 싱글톤 빈은 생성해서 주입이 가능하지만 , request 스코프 빈은 아직 생성되지 않는다 . 이 빈은 실제 고객의 요청이 와야 생성할 수 있다 !

→ Provider로 해결 가능!

여기가 핵심이다. proxyMode = ScopedProxyMode.TARGET_CLASS 를 추가해주자.
적용 대상이 인터페이스가 아닌 클래스면 TARGET_CLASS 를 선택
적용 대상이 인터페이스면 INTERFACES 를 선택
이렇게 하면 MyLogger의 가짜 프록시 클래스를 만들어두고 HTTP request와 상관 없이 가짜 프록시 클래스를 다른 빈에 미리 주입해 둘 수 있다.