싱글톤 빈 스코프와 프로토타입 빈 스코프 구현하기

미션	[백엔드] 스프링 핵심 원리 - 기본
상태	완료
설명	Spring 프레임워크에서 제공하는 싱글톤 빈 스코프와 프로토타입 빈 스코프를 구현합니다. 각 스코프의 특성을 이해하고, 이를 코드에 적용합니다. 각 스코프의 빈 동작 방식이 올바르게 구현되었는지 확인하기 위해 실행 결과를 확인합니다. 결과물로 각 스코프의 빈 동작 방식이 나타난 스크린샷을 제출합니다.

빈 스코프

> 빈이 존재할 수 있는 범위

- **싱글톤** : 기본 스코프, 스프링 컨테이너의 시작과 종료까지 유지되는 가장 넓은 범위의 스코프
- **프로토타입** : 스프링 컨테이너는 프로토타입 빈의 생성과 의존관계 주입까지만 관여하고 더는 관리하지 않는 매우 짧은 범위의 스코프
- **웹 관련**
  - **request** : 웹 요청이 들어오고 나갈때 까지 유지되는 스코프
  - **session** : 웹 세션이 생성되고 종료될 때 까지 유지되는 스코프
  - **application** : 웹의 서블릿 컨텍스트와 같은 범위로 유지되는 스코프

컴포넌트 스캔 자동 등록

```
@Scope("prototype")
@Component
public class HelloBean {}
```

```
@Scope("prototype")
@Bean
PrototypeBean HelloBean() {
    return new HelloBean();
}
```

프로토타입 스코프

싱글톤 스코프의 빈 → 항상 같은 인스턴스의 스프링 빈 반환

프로토타입 스코프 → 항상 새로운 인스턴스를 생성해서 반환

→ 스프링 컨테이너는 프로토타입 빈을 생성하고 의존관계 주입, 초기화 까지만 처리한다. 클라이언트에 빈을 반환하고 이후 스프링 컨테이너는 생성된 프로토타입 빈을 관리하지 않는다. 관리 책임은 클라이언트에게 있다.

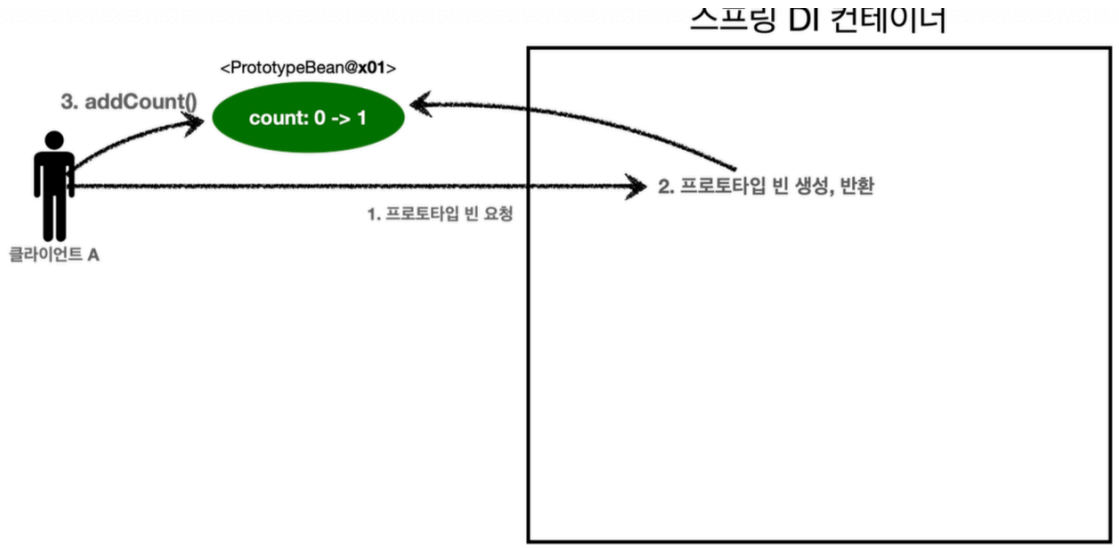
프로토타입 빈의 특징

- 스프링 컨테이너에 요청할 때 마다 새로 생성
- 스프링 컨테이너는 프로토타입 빈의 생성, 의존관계 주입, 초기화 까지만 관여한다.
- 종료 메서드가 호출되지 않는다.
- 클라이언트가 관리해야 한다.

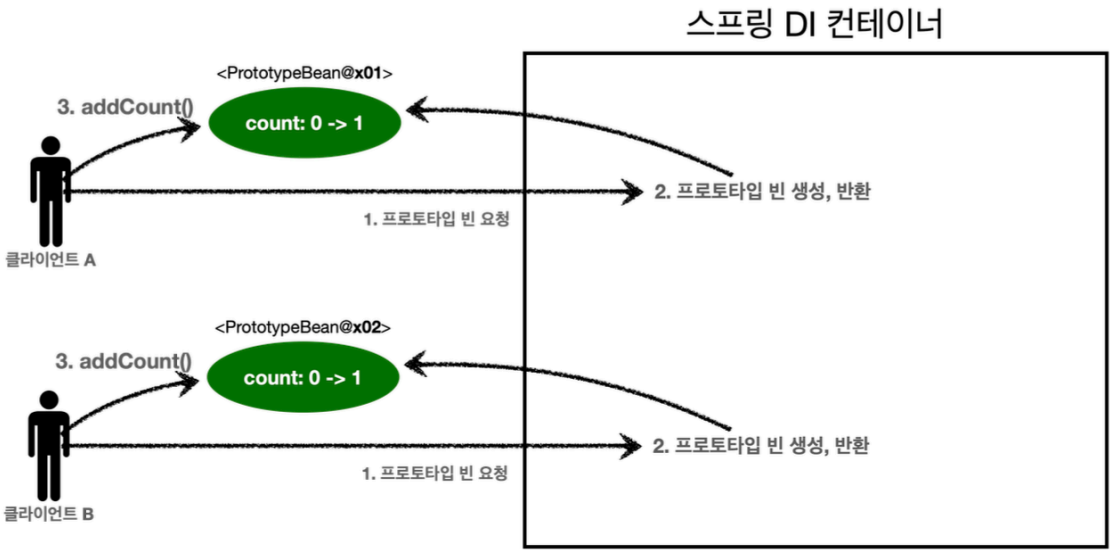
프로토타입 스코프 - 싱글톤 빈과 함께 사용시 문제점

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

프로토타입 빈 직접 요청



1. 클라이언트 A가 스프링 컨테이너에 프로토타입 빈을 요청한다.
2. 스프링 컨테이너는 프로토타입 빈을 새로 생성해서 반환(0x1)한다. 해당 빈의 count 필드 값은 0이다.
3. 클라이언트는 조회한 프로토타입 빈에 addCount()를 호출하면서 count 필드를 +1 한다. 결과적으로 프로토타입 빈(0x1)의 count는 1이 된다.



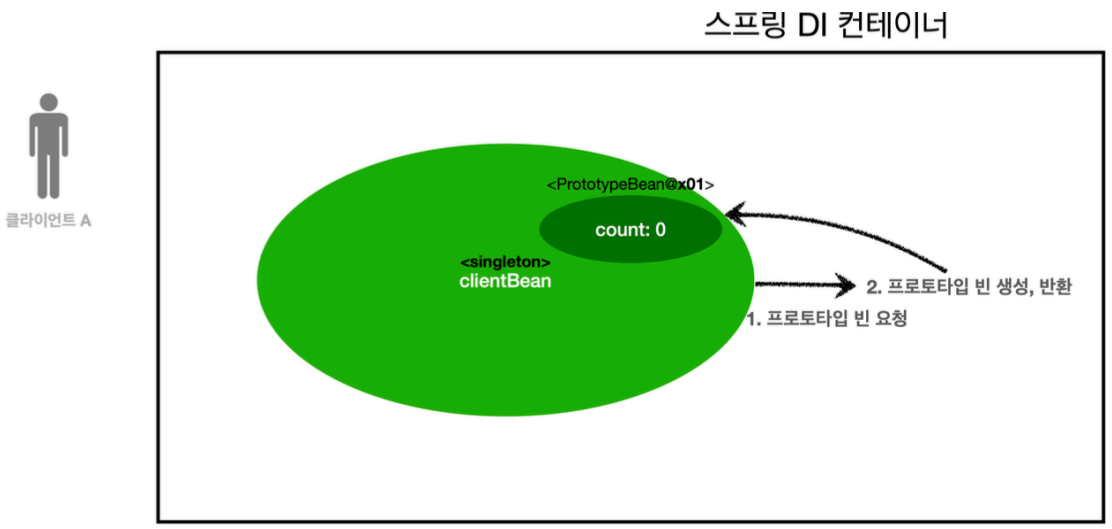
1. 클라이언트 B는 스프링 컨테이너에 프로토타입 빈을 요청한다.
2. 스프링 컨테이너는 프로토타입 빈을 새로 생성해서 반환(0x2)한다. 해당 빈의 count필드 값은 0이다.
3. 클라이언트는 조회한 프로토타입 빈에 addCount()를 호출하면서 count 필드를 +1 한다. 결과적으로 프로토타입 빈(0x2)의 count는 1이 된다.

테스트 코드

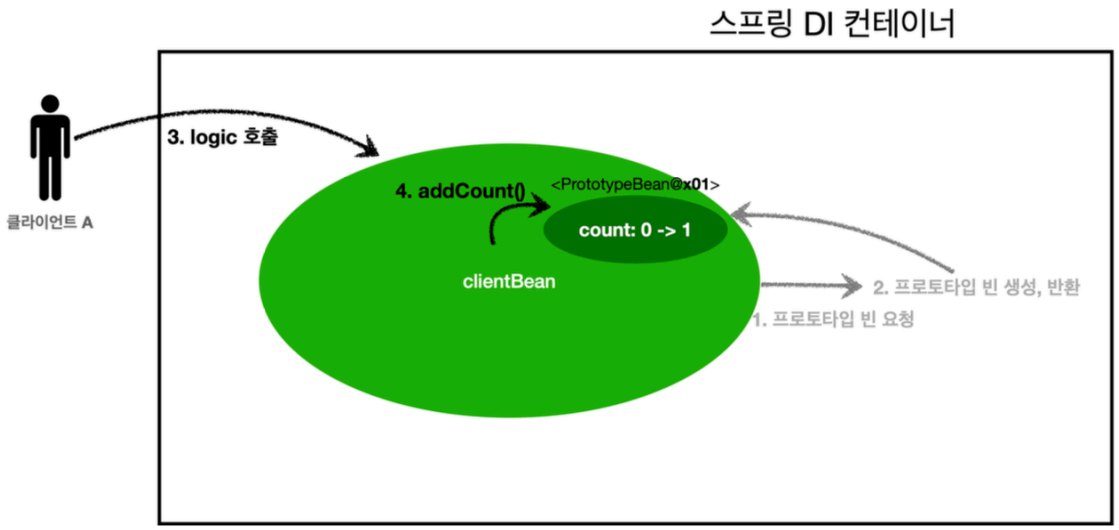
```
public class singletonWithPrototypeTest1 {

    @Test
    void prototypeFind() {
        AnnotationConfigApplicationContext ac = new AnnotationConfigApplic
ationContext(PrototypeBean.class);
        PrototypeBean prototypeBean1 = ac.getBean(PrototypeBean.class);
        PrototypeBean prototypeBean2 = ac.getBean(PrototypeBean.class);

        prototypeBean1.addCount();
        prototypeBean2.addCount();

        Assertions.assertThat(prototypeBean1.getCount()).isEqualTo(1);
        Assertions.assertThat(prototypeBean2.getCount()).isEqualTo(1);

    }

    @Scope("prototype")
    static class PrototypeBean {

        private int count = 0;

        public void addCount() {
            count++;
        }

        public int getCount() {
            return count;
        }

        @PostConstruct
        public void init() {
            System.out.println("PrototypeBean.init" + this);
        }
```

```
        @PreDestroy
        public void destroy() {
            System.out.println("PrototypeBean.destroy");
        }
    }
}
```

싱글톤 빈에서 프로토타입 빈 사용

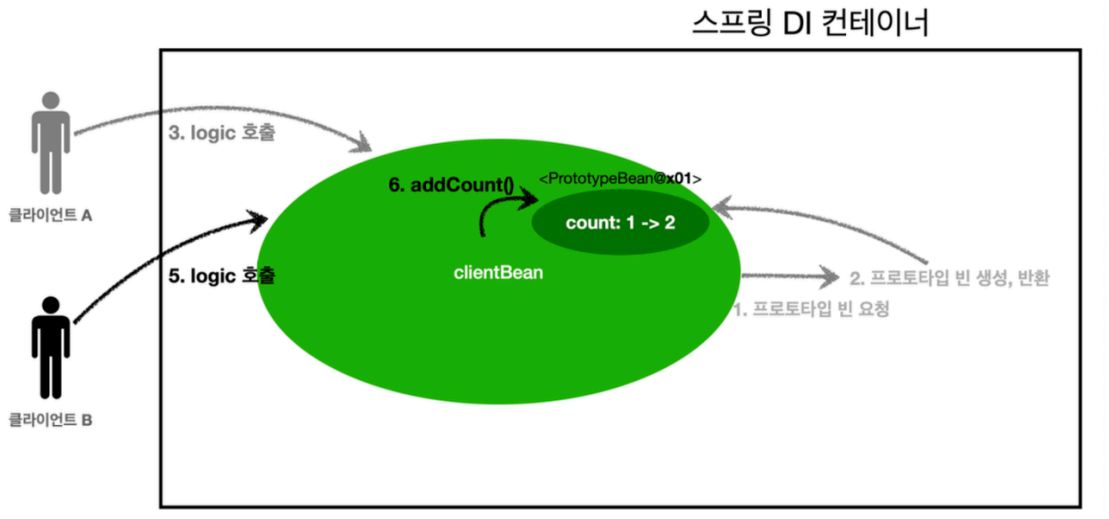


1. clientBean은 싱글톤, 스프링 컨테이너 생성 시점에 함께 생성되고 의존관계 주입도 발생한다. 주입 시점에 스프링 컨테이너에 프로토타입 빈을 요청한다.
2. 스프링 컨테이너는 프로토타입 빈을 생성해서 clientBean에 반환한다. 프로토타입 빈의 count 필드 값은 0이다. clientBean은 프로토타입 빈을 내부 필드에 보관한다. (정확히는 참조값을 보관)



클라이언트 A는 clientBean을 스프링 컨테이너에 요청해서 받는다. 싱글톤이므로 항상 같은 clientBean이 반환된다.

3. 클라이언트 A는 clientBean.logic()을 호출한다.
4. clientBean은 프로토타입 빈의 addCount()를 호출, count 증가, count 값은 1이된다.



클라이언트 B는 clientBean을 스프링 컨테이너에 요청해서 받는다. 싱글톤이므로 역시가 같은 clientBean이 반환된다.

이때 clientBean이 내부에 가지고있는 프로토타입 빈은 이미 과거에 주입이 끝난 빈이다. 주입 시점에 스프링 컨테이너에 요청해서 프로토타입 빈이 생성되었고, 사용할때마다 새로 생성되는 것은 아니다.

5. 클라이언트 B는 clientBean.logic()을 호출한다.

6. clientBean은 프로토타입 빈의 addCount()를 호출, count 값을 증가한다. count 값은 2가 된다.

테스트 코드

```
public class singletonWithPrototypeTest1 {

    @Test
    void singletonClientUsePrototype() {
        AnnotationConfigApplicationContext ac = new AnnotationConfigApplic
ationContext(ClientBean.class, PrototypeBean.class);
        ClientBean clientBean1 = ac.getBean(ClientBean.class);
        int count1 = clientBean1.logic();
        Assertions.assertThat(count1).isEqualTo(1);
        ClientBean clientBean2 = ac.getBean(ClientBean.class);
        int count2 = clientBean2.logic();
        Assertions.assertThat(count2).isEqualTo(2);
    }

    @Scope("prototype")
    static class PrototypeBean {

        private int count = 0;

        public void addCount() {
            count++;
        }

        public int getCount() {
            return count;
        }

        @PostConstruct
        public void init() {
            System.out.println("PrototypeBean.init" + this);
        }

        @PreDestroy
```

```
        public void destroy() {
            System.out.println("PrototypeBean.destroy");
        }
    }

    static class ClientBean {
        private final PrototypeBean prototypeBean;

        @Autowired
        public ClientBean(PrototypeBean prototypeBean) {
            this.prototypeBean = prototypeBean;
        }

        public int logic() {
            prototypeBean.addCount();
            return prototypeBean.getCount();
        }
    }
}
```

문제점

스프링은 일반적으로 싱글톤 빈을 사용, 싱글톤 빈이 프로토타입 빈을 사용하게 된다. 싱글톤 빈은 생성 시점에만 의존관계 주입을 받기 때문에 프로토타입 빈이 새로 생성되지만, 싱글톤 빈과 함께 계속 유지되는 것이 문제다.

→ 사용할 때 마다 새로 생성해서 사용하도록 하고싶다.

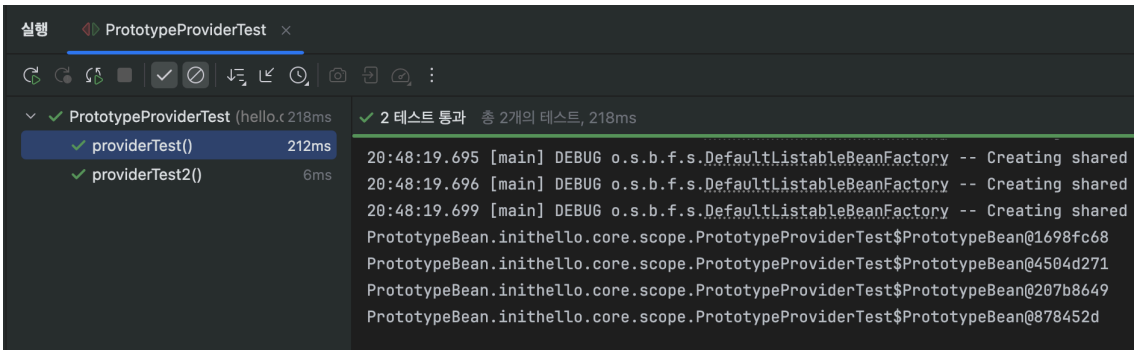
Provider

> 싱글톤 빈과 프로토타입 빈을 함께 사용할 때, 항상 새로운 프로토타입 빈을 생성하도록 하자.

스프링 컨테이너에 요청

```
public class PrototypeProviderTest {

    @Test
    void providerTest() {
        AnnotationConfigApplicationContext ac = new AnnotationConfigApplic
ationContext(ClientBean.class, PrototypeBean.class);

        ClientBean clientBean1 = ac.getBean(ClientBean.class);
        ClientBean clientBean2 = ac.getBean(ClientBean.class);

        clientBean1.logic();
        clientBean2.logic();

        Assertions.assertThat(clientBean1.logic()).isEqualTo(1);
        Assertions.assertThat(clientBean2.logic()).isEqualTo(1);
    }

    static class ClientBean {

        @Autowired
        private ApplicationContext ac;

        public int logic() {
            PrototypeBean prototypeBean = ac.getBean(PrototypeBean.class);
            prototypeBean.addCount();
            return prototypeBean.getCount();
        }
    }

    @Scope("prototype")
    static class PrototypeBean {

        private int count = 0;

        public void addCount() {
            count++;
        }
```

```
        public int getCount() {
            return count;
        }

        @PostConstruct
        public void init() {
            System.out.println("PrototypeBean.init" + this);
        }

        @PreDestroy
        public void destroy() {
            System.out.println("PrototypeBean.destroy");
        }
    }

}
```



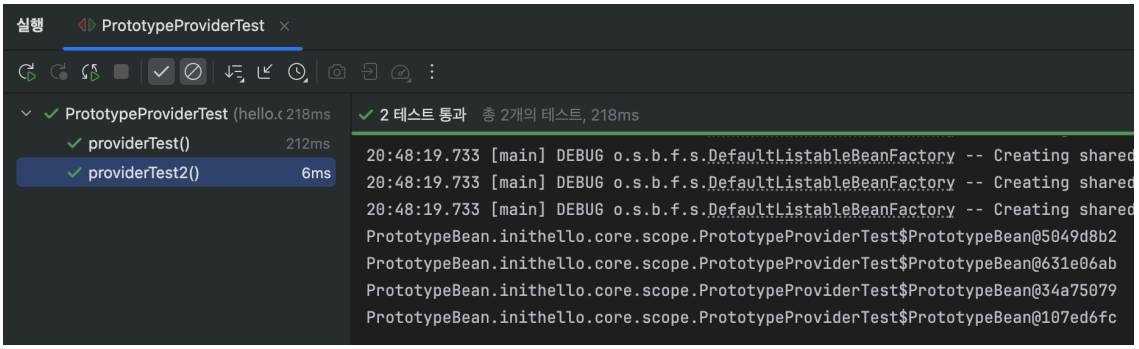
항상 새로운 프로토타입 빈 생성 되는것을 볼 수 있다.

- `ac.getBean()`을 통해 항상 새로운 프로토타입 빈이 생성
- 의존관계를 외부에서 주입(DI)받는 것이 아닌, 직접 필요한 의존관계를 찾는 것을 Dependency Lookup(DL) 의존관계 조회라 한다.
- 하지만 지금처럼 스프링 애플리케이션 컨텍스트 전체를 주입받게 되면 스프링 컨테이너에 종속적인 코드가 되고, 단위 테스트도 어려워 진다.
- 따라서 DL의 기능만을 제공하는 무언가가 필요

ObjectFactory, ObjectProvider

지정한 빈을 컨테이너에서 대신 찾아주는 DL 서비스를 제공하는 것이 바로 `ObjectProvider`이다.

```
static class ClientBean2 {

    @Autowired
    private ObjectProvider<PrototypeBean> prototypeBeanProvider;

    public int logic() {
        PrototypeBean prototypeBean = prototypeBeanProvider.getObject();
        prototypeBean.addCount();
        int count = prototypeBean.getCount();
        return count;
    }
}
```



- `prototypeBeanProvider.getObject()`를 통해 역시나 항상 새로운 프로토타입 빈이 생성되는 것을 확인할 수 있다.
- `ObjectProvider`의 `getObject()`를 호출하면 내부에서는 스프링 컨테이너를 통해 해당 빈을 찾아서 반환한다. (DL)

정리

- 프로토타입 빈은 언제 사용할까? → 매번 사용할 때 마다 의존관계 주입이 완료된 새로운 객체가 필요할 때 사용하면 된다.
- `ObjectProvider`, `JSR330 Provider` 등은 프로토타입 뿐만 아니라 DL이 필요한 경우 언제든지 사용 가능