정리

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

9. 빈 스코프

#인강/2.스프링 핵심원리 - 기본편/기본편#

목차

빈 스코프란?

지금까지 우리는 스프링 빈이 스프링 컨테이너의 시작과 함께 생성되어서 스프링 컨테이너가 종료될 때까지 유지된다고 학습했다. 이것은 스프링 빈이 기본적으로 싱글톤 스코프로 생성되기 때문이다. 스코프는 번역 그대로 빈이 존재할 수 있는 범위를 뜻한다.

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.

싱글톤,프로토,request정도만 알면됨

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

빈 스코프는 다음과 같이 지정할 수 있다.

컴포넌트 스캔 자동 등록

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

수동 등록

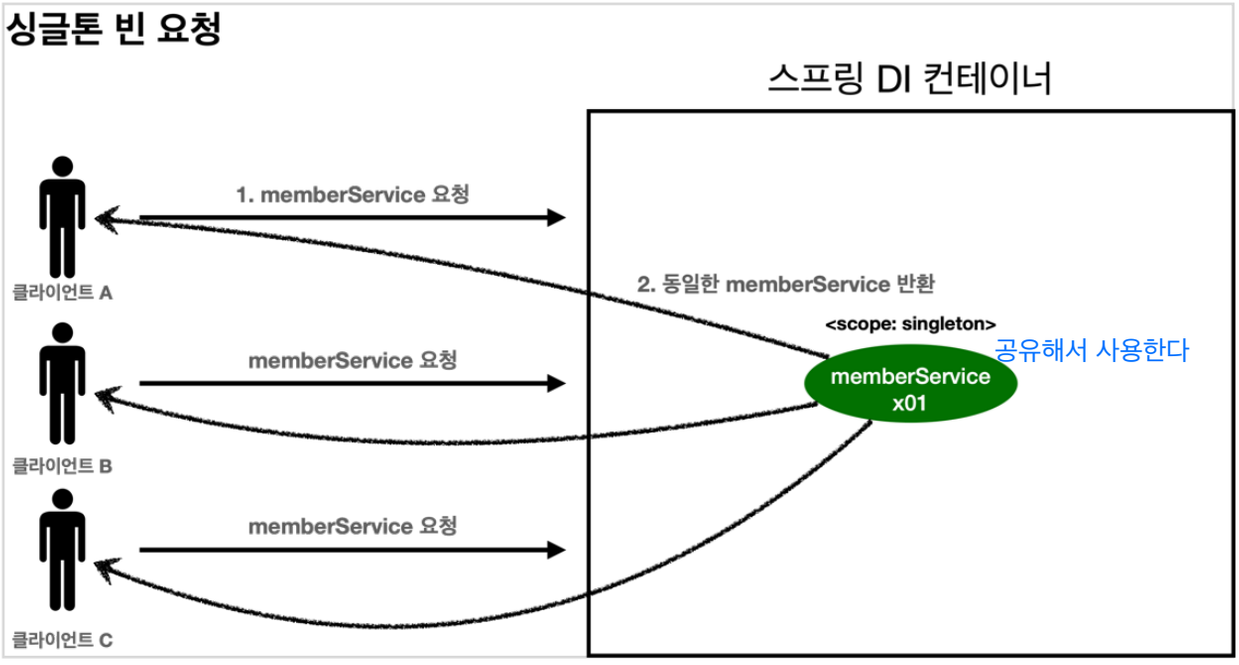
```
@Scope("prototype")
@Bean
PrototypeBean HelloBean() {
    return new HelloBean();
}
```

지금까지 싱글톤 스코프를 계속 사용해보았으니, 프로토타입 스코프부터 확인해보자.

프로토타입 스코프

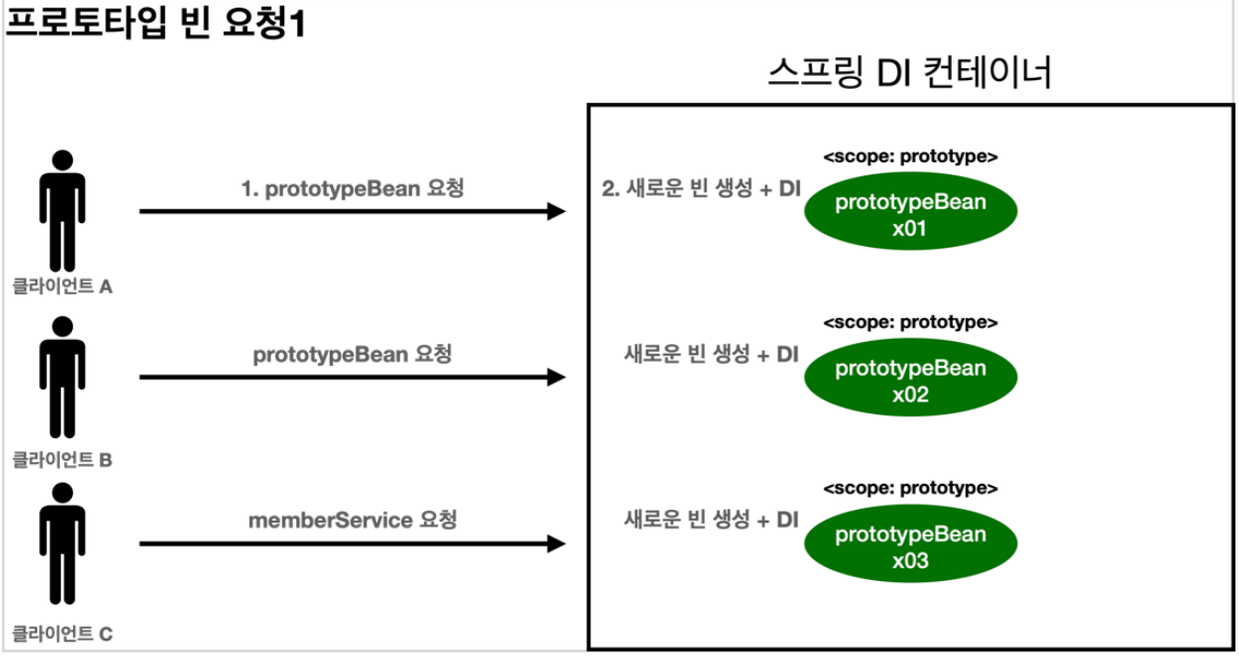
싱글톤 스코프의 빈을 조회하면 스프링 컨테이너는 항상 같은 인스턴스의 스프링 빈을 반환한다. 반면에 프로토타입 스코프를 스프링 컨테이너에 조회하면 스프링 컨테이너는 항상 새로운 인스턴스를 생성해서 반환한다.

싱글톤 빈 요청



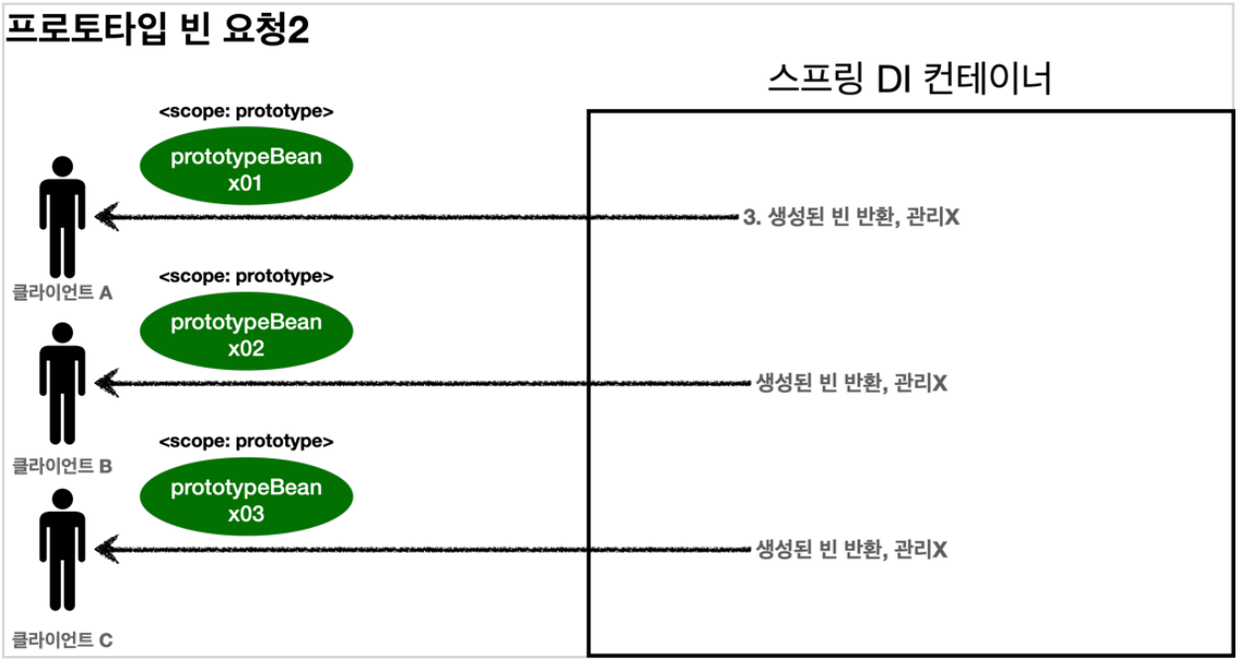
- 1. 싱글톤 스코프의 빈을 스프링 컨테이너에 요청한다.
- 2. 스프링 컨테이너는 본인이 관리하는 스프링 빈을 반환한다.
- 3. 이후에 스프링 컨테이너에 같은 요청이 와도 같은 객체 인스턴스의 스프링 빈을 반환한다.

프로토타입 빈 요청1



- 1. 프로토타입 스코프의 빈을 스프링 컨테이너에 요청한다.
- 2. 스프링 컨테이너는 이 시점에 프로토타입 빈을 생성하고, 필요한 의존관계를 주입한다.

프로토타입 빈 요청2 간단 - 생성하는 주기만함



- 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.

이것이 중요!

코드로 확인해보자.

싱글톤 스코프 빈 테스트

```
package hello.core.scope;

import org.junit.jupiter.api.Test;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Scope;

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

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

public class SingletonTest {
```

```
    @Test
    public void singletonBeanFind() {

        AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(SingletonBean.class);
        SingletonBean singletonBean1 = ac.getBean(SingletonBean.class);
        SingletonBean singletonBean2 = ac.getBean(SingletonBean.class);
        System.out.println("singletonBean1 = " + singletonBean1);
        System.out.println("singletonBean2 = " + singletonBean2);
        assertThat(singletonBean1).isSameAs(singletonBean2);

        ac.close(); //종료
    }

    @Scope("singleton")
    static class SingletonBean {

        @PostConstruct
        public void init() {
            System.out.println("SingletonBean.init");
        }

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

}
```

@Configuration을 하지않고 직접 클래스를 등록하면 클래스가 componentscan한 효과를 가져옴

먼저 싱글톤 스코프의 빈을 조회하는 `singletonBeanFind()` 테스트를 실행해보자.

실행 결과

```
SingletonBean.init
singletonBean1 = hello.core.scope.PrototypeTest$SingletonBean@54504ecd
singletonBean2 = hello.core.scope.PrototypeTest$SingletonBean@54504ecd
```

```
org.springframework.context.annotation.AnnotationConfigApplicationContext -
Closing SingletonBean.destroy
```

- 빈 초기화 메서드를 실행하고,
- 같은 인스턴스의 빈을 조회하고,
- 종료 메서드까지 정상 호출 된 것을 확인할 수 있다.

프로토타입 스코프 빈 테스트

```
package hello.core.scope;

import org.junit.jupiter.api.Test;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;

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

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

public class PrototypeTest {

    @Test
    public void prototypeBeanFind() {

        AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(PrototypeBean.class);
        System.out.println("find prototypeBean1");
        PrototypeBean prototypeBean1 = ac.getBean(PrototypeBean.class);
        System.out.println("find prototypeBean2");
        PrototypeBean prototypeBean2 = ac.getBean(PrototypeBean.class);
        System.out.println("prototypeBean1 = " + prototypeBean1);
        System.out.println("prototypeBean2 = " + prototypeBean2);
        assertThat(prototypeBean1).isNotSameAs(prototypeBean2);
        ac.close(); //종료
```

```
    }

    @Scope("prototype")
    static class PrototypeBean {

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

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

프로토타입 스코프의 빈을 조회하는 `prototypeBeanFind()` 테스트를 실행해보자.

실행 결과

```
find prototypeBean1
PrototypeBean.init
find prototypeBean2
PrototypeBean.init
prototypeBean1 = hello.core.scope.PrototypeTest$PrototypeBean@13d4992d
prototypeBean2 = hello.core.scope.PrototypeTest$PrototypeBean@302f7971
org.springframework.context.annotation.AnnotationConfigApplicationContext -
Closing
```

- 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.

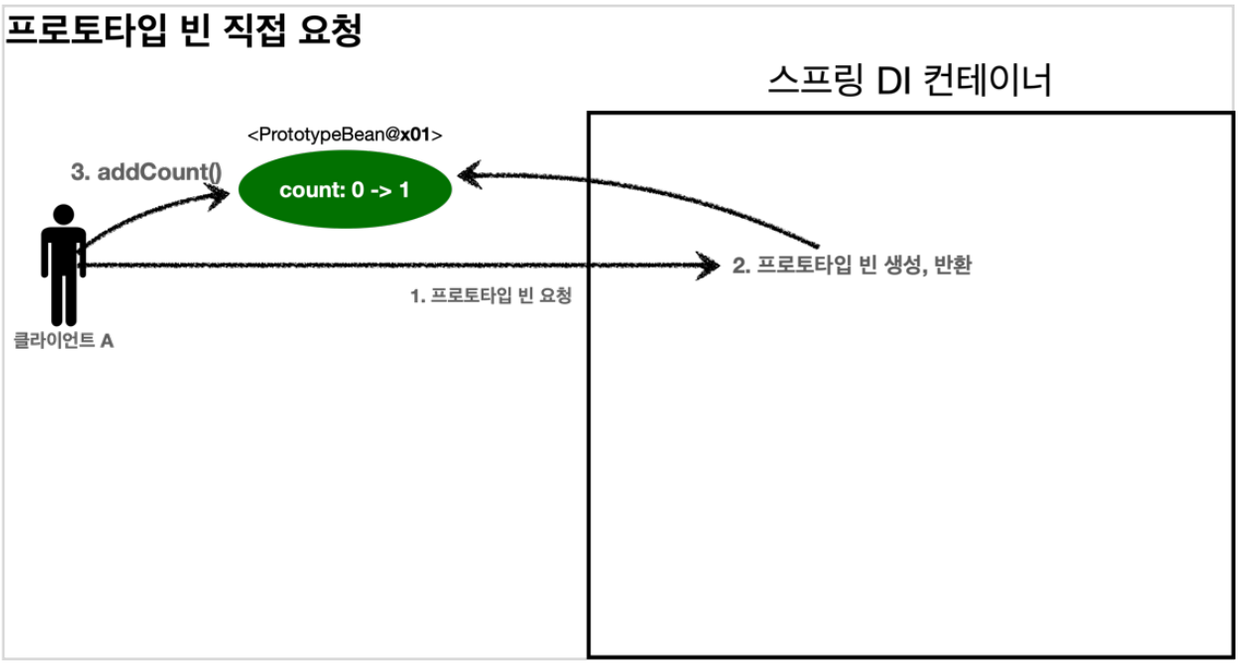
프로토타입 스코프 - 싱글톤 빈과 함께 사용시 문제점

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. 그림과 코드로 설명하겠다.

먼저 스프링 컨테이너에 프로토타입 빈을 직접 요청하는 예제를 보자.

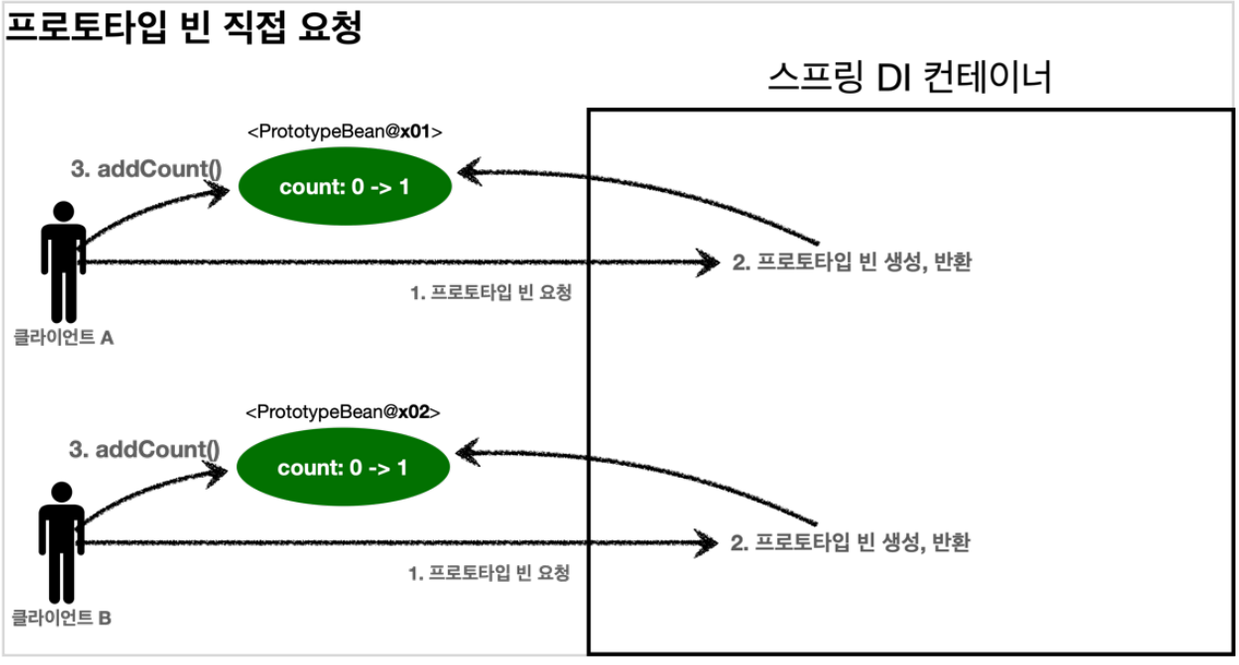
프로토타입 빈 직접 요청

스프링 컨테이너에 프로토타입 빈 직접 요청1



- 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



- 1. 클라이언트B는 스프링 컨테이너에 프로토타입 빈을 요청한다.
- 2. 스프링 컨테이너는 프로토타입 빈을 새로 생성해서 반환(**x02**)한다. 해당 빈의 count 필드 값은 0이다.
- 3. 클라이언트는 조회한 프로토타입 빈에 `addCount()` 를 호출하면서 count 필드를 +1 한다.
- 결과적으로 프로토타입 빈(**x02**)의 count는 1이 된다.

코드로 확인

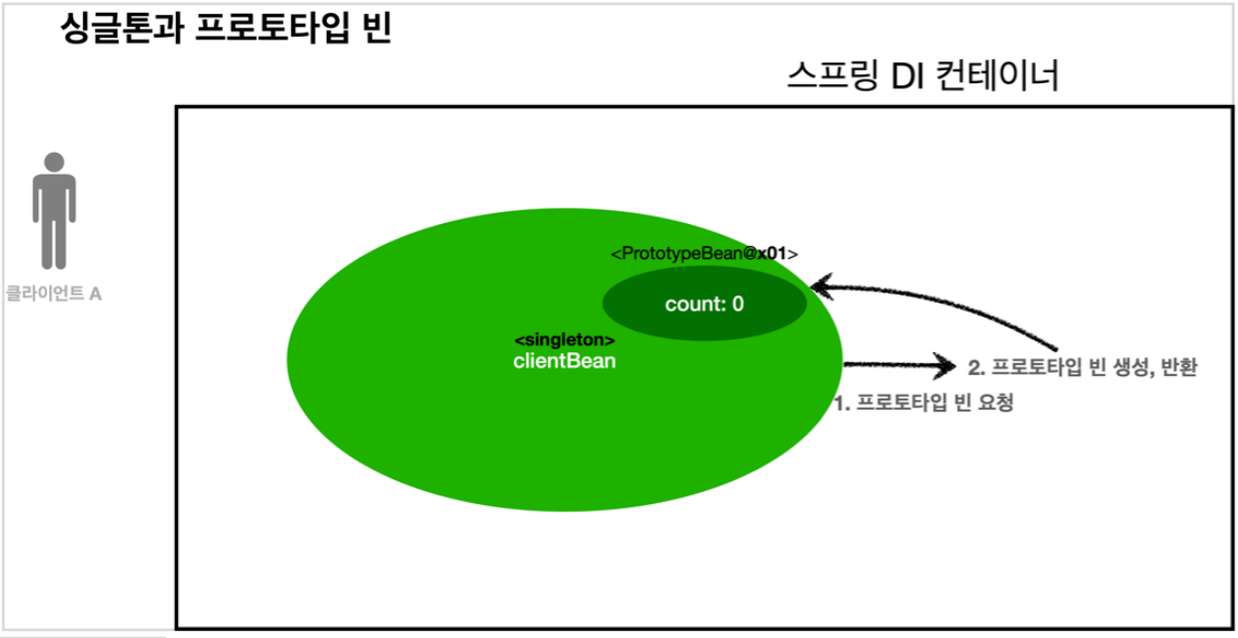
```
public class SingletonWithPrototypeTest1 {

    @Test
    void prototypeFind() {
        AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(PrototypeBean.class);
        PrototypeBean prototypeBean1 = ac.getBean(PrototypeBean.class);
        prototypeBean1.addCount();
        assertThat(prototypeBean1.getCount()).isEqualTo(1);

        PrototypeBean prototypeBean2 = ac.getBean(PrototypeBean.class);
        prototypeBean2.addCount();
```

```
        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

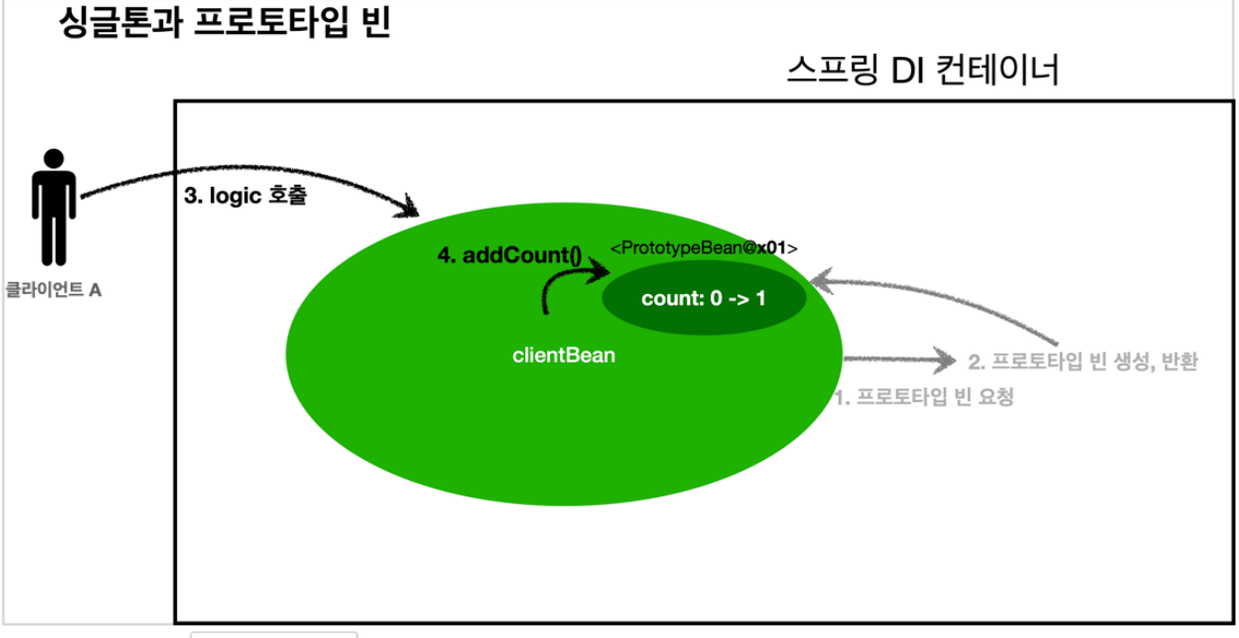
이번에는 `clientBean` 이라는 싱글톤 빈이 의존관계 주입을 통해서 프로토타입 빈을 주입받아서 사용하는 예를 보자.

싱글톤에서 프로토타입 빈 사용1



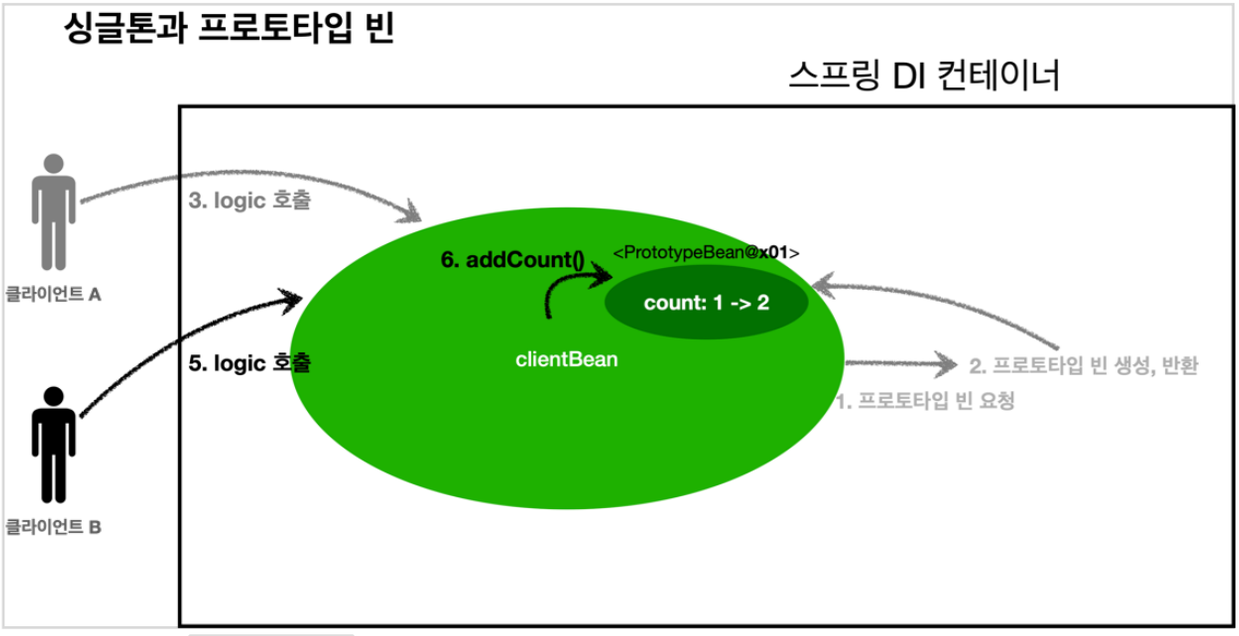
- `clientBean`은 싱글톤이므로, 보통 스프링 컨테이너 생성 시점에 함께 생성되고, 의존관계 주입도 발생한다.
- 1. `clientBean`은 의존관계 자동 주입을 사용한다. 주입 시점에 스프링 컨테이너에 프로토타입 빈을 요청한다.
- 2. 스프링 컨테이너는 프로토타입 빈을 생성해서 `clientBean`에 반환한다. 프로토타입 빈의 count 필드 값은 0이다.
- 이제 `clientBean`은 프로토타입 빈을 내부 필드에 보관한다. (정확히는 참조값을 보관한다.)

싱글톤에서 프로토타입 빈 사용2



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

싱글톤에서 프로토타입 빈 사용3



- 클라이언트 B는 `clientBean`을 스프링 컨테이너에 요청해서 받는다.싱글톤이므로 항상 같은 `clientBean`이 반환된다.
- **여기서 중요한 점이 있는데, clientBean이 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 prototypeBean의 `addCount()`를 호출해서 프로토타입 빈의 count를 증가한다. 원래 count 값이 1이었으므로 2가 된다.

싱글톤이 프로토를 먹어버려서 프로토타입이 계속생성되는것이아니라 먹혀있는 프로토가 계속 사용된다.

테스트 코드

```
package hello.core.scope;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Scope;

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

import static org.assertj.core.api.Assertions.*;
```

```
public class SingletonWithPrototypeTest1 {
                                                @Scope의 디폴트는 싱글톤임
    @Test
    void singletonClientUsePrototype() {
        AnnotationConfigApplicationContext ac = new
AnnotationConfigApplicationContext(ClientBean.class, PrototypeBean.class);

        ClientBean clientBean1 = ac.getBean(ClientBean.class);
        int count1 = clientBean1.logic();
        assertThat(count1).isEqualTo(1);

        ClientBean clientBean2 = ac.getBean(ClientBean.class);
        int count2 = clientBean2.logic();
        assertThat(count2).isEqualTo(2);
    }

    static class ClientBean {

        private final PrototypeBean prototypeBean;

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

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

    @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");
        }
    }

}
```

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 생성**된다. 예를 들어서 clientA, clientB가 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.

프로토타입 스코프 - 싱글톤 빈과 함께 사용시 Provider로 문제 해결

싱글톤 빈과 프로토타입 빈을 함께 사용할 때, 어떻게 하면 사용할 때 마다 항상 새로운 프로토타입 빈을 생성할 수 있을까?

스프링 컨테이너에 요청

가장 간단한 방법은 싱글톤 빈이 프로토타입을 사용할 때 마다 스프링 컨테이너에 새로 요청하는 것이다.

```
public class PrototypeProviderTest {

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

        ClientBean clientBean1 = ac.getBean(ClientBean.class);
        int count1 = clientBean1.logic();
        assertThat(count1).isEqualTo(1);

        ClientBean clientBean2 = ac.getBean(ClientBean.class);
        int count2 = clientBean2.logic();
        assertThat(count2).isEqualTo(1);
    }

    static class ClientBean {

        @Autowired
        private ApplicationContext ac;

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

```
    @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");
        }
    }

}
```

핵심 코드

```
@Autowired
private ApplicationContext ac;

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

```
}
```

- 실행해보면 `ac.getBean()` 을 통해서 항상 새로운 프로토타입 빈이 생성되는 것을 확인할 수 있다.
- 의존관계를 외부에서 주입(DI) 받는게 아니라 이렇게 직접 필요한 의존관계를 찾는 것을 Dependency Lookup (DL) 의존관계 조회(탐색) 이라한다.
- 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

Provider가 기능이 더 많음

지정한 빈을 컨테이너에서 대신 찾아주는 DL 서비스를 제공하는 것이 바로 `ObjectProvider` 이다. 참고로 과거에는 `ObjectFactory` 가 있었는데, 여기에 편의 기능을 추가해서 `ObjectProvider` 가 만들어졌다.

```
@Autowired
private ObjectProvider<PrototypeBean> prototypeBeanProvider;

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

- 실행해보면 `prototypeBeanProvider.getObject()` 을 통해서 항상 새로운 프로토타입 빈이 생성되는 것을 확인할 수 있다.
- `ObjectProvider` 의 `getObject()` 를 호출하면 내부에서는 스프링 컨테이너를 통해 해당 빈을 찾아서 반환한다. (**DL**)
- 스프링이 제공하는 기능을 사용하지만, 기능이 단순하므로 단위테스트를 만들거나 mock 코드를 만들기는 훨씬 쉬워진다.
- `ObjectProvider` 는 지금 딱 필요한 DL 정도의 기능만 제공한다.

특징

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

JSR-330 Provider 스프링에 의존하지 않음

마지막 방법은 `javax.inject.Provider` 라는 JSR-330 자바 표준을 사용하는 방법이다.
이 방법을 사용하려면 `javax.inject:javax.inject:1` 라이브러리를 gradle에 추가해야 한다.

javax.inject.Provider 참고용 코드

```
package javax.inject;
public interface Provider<T> {
    T get();
}
```

```
//implementation 'javax.inject:javax.inject:1' gradle 추가 필수
@Autowired
private Provider<PrototypeBean> provider;

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

- 실행해보면 `provider.get()` 을 통해서 항상 새로운 프로토타입 빈이 생성되는 것을 확인할 수 있다.
- `provider` 의 `get()` 을 호출하면 내부에서는 스프링 컨테이너를 통해 해당 빈을 찾아서 반환한다. (**DL**)
- 자바 표준이고, 기능이 단순하므로 단위테스트를 만들거나 mock 코드를 만들기는 훨씬 쉬워진다.
- `Provider` 는 지금 딱 필요한 DL 정도의 기능만 제공한다.

특징

- `get()` 메서드 하나로 기능이 매우 단순하다.
- 별도의 라이브러리가 필요하다.
- 자바 표준이므로 스프링이 아닌 다른 컨테이너에서도 사용할 수 있다.

사실 거의 사용할 일은 없다

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 Provider를 사용할 것인지, 아니면 스프링이 제공하는 ObjectProvider를 사용할 것인지 고민이 될 것이다. ObjectProvider는 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 Provider를 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

지금까지 싱글톤과 프로토타입 스코프를 학습했다. 싱글톤은 스프링 컨테이너의 시작과 끝까지 함께하는 매우 긴 스코프이고, 프로토타입은 생성과 의존관계 주입, 그리고 초기화까지만 진행하는 특별한 스코프이다.

이번에는 웹 스코프에 대해서 알아보자

싱글톤-스프링의 시작과 끝
프로토-의존관계주입 후 초기화하고 끝

웹 스코프의 특징

- 웹 스코프는 웹 환경에서만 동작한다.
- 웹 스코프는 프로토타입과 다르게 스프링이 해당 스코프의 종료시점까지 관리한다. 따라서 종료 메서드가 호출된다.

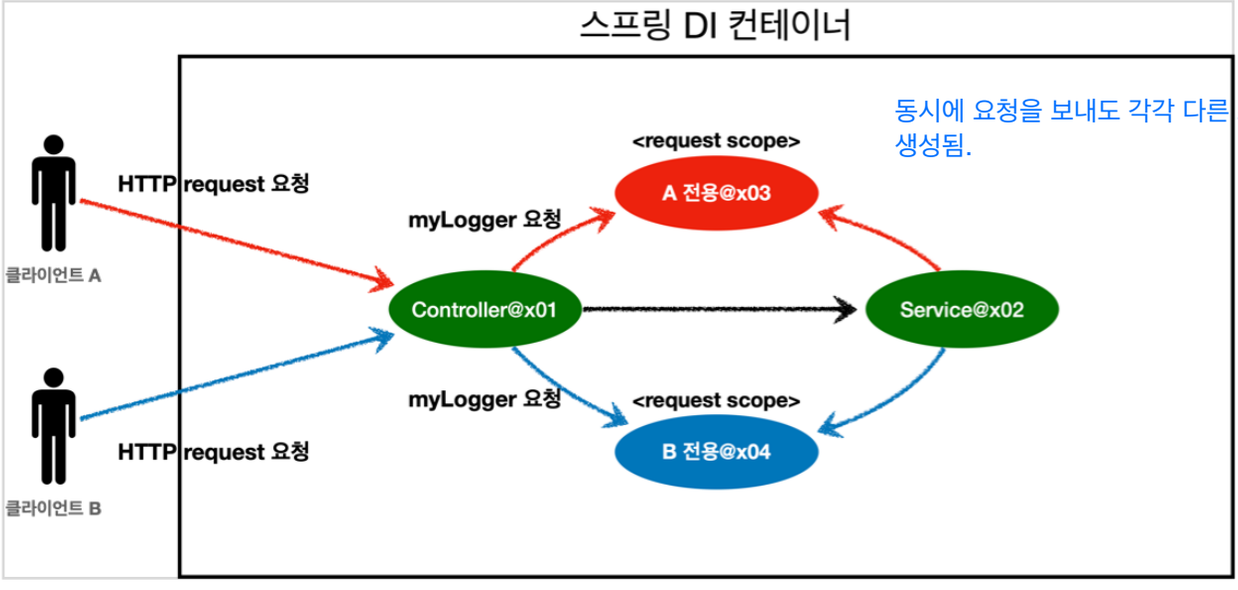
웹 스코프 종류

- **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

요청마다 각각 다른 스코프임

사실 세션이나, 서블릿 컨텍스트, 웹 소켓 같은 용어를 잘 모르는 분들도 있을 것이다. 여기서는 request 스코프를 예제로 설명하겠다. 나머지도 범위만 다르지 동작 방식은 비슷하다.

HTTP request 요청 당 각각 할당되는 request 스코프



동시에 요청을 보내도 각각 다른 스프링 빈이 생성됨.

각각 할당되어서 요청,응답동안의 싸이클동안 은 같은 스코프를 사용함.

request 스코프 예제 만들기

웹 환경 추가

웹 스코프는 웹 환경에서만 동작하므로 web 환경이 동작하도록 라이브러리를 추가하자.

build.gradle에 추가

```
//web 라이브러리 추가
implementation 'org.springframework.boot:spring-boot-starter-web'
```

이제 `hello.core.CoreApplication`의 main 메서드를 실행하면 웹 애플리케이션이 실행되는 것을 확인할 수 있다.

```
Tomcat started on port(s): 8080 (http) with context path ''
Started CoreApplication in 0.914 seconds (JVM running for 1.528)
```

참고: `spring-boot-starter-web` 라이브러리를 추가하면 스프링 부트는 내장 톰캣 서버를 활용해서 웹 서버와 스프링을 함께 실행시킨다.

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

만약 기본 포트인 8080 포트를 다른곳에서 사용중이어서 오류가 발생하면 포트를 변경해야 한다. 9090 포트로 변경하려면 다음 설정을 추가하자.
`main/resources/application.properties`

```
server.port=9090
```

request 스코프 예제 개발

동시에 여러 HTTP 요청이 오면 정확히 어떤 요청이 남긴 로그인지 구분하기 어렵다.
이럴때 사용하기 딱 좋은것이 바로 request 스코프이다.

다음과 같이 로그가 남도록 request 스코프를 활용해서 추가 기능을 개발해보자.

```
[d06b992f...] request scope bean create
```

이 url을 요청했구나!

```
[d06b992f...][http://localhost:8080/log-demo] controller test
[d06b992f...][http://localhost:8080/log-demo] service id = testId
[d06b992f...] request scope bean close
```

- 기대하는 공통 포멧: [UUID][requestURL]{message}
- UUID를 사용해서 HTTP 요청을 구분하자.
- requestURL 정보도 추가로 넣어서 어떤 URL을 요청해서 남은 로그인지 확인하자.

먼저 코드로 확인해보자.

MyLogger

```
package hello.core.common;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.util.UUID;

@Component
@Scope(value = "request")
public class MyLogger {

    private String uuid;
    private String requestURL;

    public void setRequestURL(String requestURL) {
        this.requestURL = requestURL;
    }

    public void log(String message) {
        System.out.println("[" + uuid + "]" + "[" + requestURL + "] " +
message);
    }

    @PostConstruct
```

```
    public void init() {
        uuid = UUID.randomUUID().toString(); <-유니크한 아이디가 생성됨
        System.out.println("[" + uuid + "] request scope bean create:" + this);
    }

    @PreDestroy     고객요청이 빠져나가면 이 메소드가 호출이 됨
    public void close() {
        System.out.println("[" + uuid + "] request scope bean close:" + this);
    }
}
```

- 로그를 출력하기 위한 `MyLogger` 클래스이다.
- `@Scope(value = "request")` 를 사용해서 request 스코프로 지정했다. 이제 이 빈은 HTTP 요청 당 하나씩 생성되고, HTTP 요청이 끝나는 시점에 소멸된다.
- 이 빈이 생성되는 시점에 자동으로 `@PostConstruct` 초기화 메서드를 사용해서 uuid를 생성해서 저장해둔다. 이 빈은 HTTP 요청 당 하나씩 생성되므로, uuid를 저장해두면 다른 HTTP 요청과 구분할 수 있다.
- 이 빈이 소멸되는 시점에 `@PreDestroy` 를 사용해서 종료 메시지를 남긴다.
- `requestURL` 은 이 빈이 생성되는 시점에는 알 수 없으므로, 외부에서 setter로 입력 받는다.

LogDemoController

```
package hello.core.web;

import hello.core.common.MyLogger;
import hello.core.logdemo.LogDemoService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.servlet.http.HttpServletRequest;

@Controller
@RequiredArgsConstructor  생성자에 Autowired로 자동주입
public class LogDemoController {
```

```
    private final LogDemoService logDemoService;
    private final MyLogger myLogger;

    @RequestMapping("log-demo")
    @ResponseBody
    public String logDemo(HttpServletRequest request) {  자바에서 제공하는 http고객정보를 받을 수 있음
        String requestURL = request.getRequestURL().toString();
        myLogger.setRequestURL(requestURL);

        myLogger.log("controller test");
        logDemoService.logic("testId");
        return "OK";
    }
}
```

- 로거가 잘 작동하는지 확인하는 테스트용 컨트롤러다.
- 여기서 HttpServletRequest를 통해서 요청 URL을 받았다.
  - requestURL 값 `http://localhost:8080/log-demo`
- 이렇게 받은 requestURL 값을 myLogger에 저장해둔다. myLogger는 HTTP 요청 당 각각 구분되므로 다른 HTTP 요청 때문에 값이 섞이는 걱정은 하지 않아도 된다.
- 컨트롤러에서 controller test라는 로그를 남긴다.

> **참고:** requestURL을 MyLogger에 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.

LogDemoService 추가

```
package hello.core.logdemo;

import hello.core.common.MyLogger;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
```

```
@Service
@RequiredArgsConstructor
public class LogDemoService {

    private final MyLogger myLogger;

    public void logic(String id) {
        myLogger.log("service id = " + id);
    }

}
```

- 비즈니스 로직이 있는 서비스 계층에서도 로그를 출력해보자.
- 여기서 중요한점이 있다. request scope를 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 scope의 MyLogger 덕분에 이런 부분을 파라미터로 넘기지 않고, MyLogger의 멤버변수에 저장해서 코드와 계층을 깔끔하게 유지할 수 있다.

이제 실행을 해보자.

기대하는 출력

```
[d06b992f...] request scope bean create
[d06b992f...][http://localhost:8080/log-demo] controller test
[d06b992f...][http://localhost:8080/log-demo] service id = testId
[d06b992f...] request scope bean close
```

실제는 기대와 다르게 애플리케이션 실행 시점에 오류 발생

```
Error creating bean with name 'myLogger': Scope 'request' is not active for the
current thread; consider defining a scoped proxy for this bean if you intend to
refer to it from a singleton;
```

스프링 애플리케이션을 실행 시키면 오류가 발생한다. 메시지 마지막에 싱글톤이라는 단어가 나오고…
스프링 애플리케이션을 실행하는 시점에 싱글톤 빈은 생성해서 주입이 가능하지만, request 스코프 빈은 아직 생성되지 않는다. 이 빈은 실제 고객의 요청이 와야 생성할 수 있다!

스코프와 Provider

첫번째 해결방안은 앞서 배운 Provider를 사용하는 것이다.
간단히 ObjectProvider를 사용해보자.

```
package hello.core.web;

import hello.core.common.MyLogger;
import hello.core.logdemo.LogDemoService;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.servlet.http.HttpServletRequest;

@Controller
@RequiredArgsConstructor
public class LogDemoController {

    private final LogDemoService logDemoService;
    private final ObjectProvider<MyLogger> myLoggerProvider;

    @RequestMapping("log-demo")
    @ResponseBody
    public String logDemo(HttpServletRequest request) {
        String requestURL = request.getRequestURL().toString();
        MyLogger myLogger = myLoggerProvider.getObject();
        myLogger.setRequestURL(requestURL);
```

```
        myLogger.log("controller test");
        logDemoService.logic("testId");
        return "OK";
    }
}
```

```
package hello.core.logdemo;

import hello.core.common.MyLogger;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class LogDemoService {

    private final ObjectProvider<MyLogger> myLoggerProvider;

    public void logic(String id) {
        MyLogger myLogger = myLoggerProvider.getObject();
        myLogger.log("service id = " + id);
    }

}
```

`main()` 메서드로 스프링을 실행하고, 웹 브라우저에 `http://localhost:8080/log-demo` 를 입력하자.

드디어 잘 작동하는 것을 확인할 수 있다.

```
[d06b992f...] request scope bean create
[d06b992f...][http://localhost:8080/log-demo] controller test
[d06b992f...][http://localhost:8080/log-demo] service id = testId
```

```
[d06b992f...] request scope bean close
```

- `ObjectProvider` 덕분에 `ObjectProvider.getObject()` 를 호출하는 시점까지 request scope **빈의 생성을 지연**할 수 있다.
- `ObjectProvider.getObject()` 를 호출하시는 시점에는 HTTP 요청이 진행중이므로 request scope 빈의 생성이 정상 처리된다.
- `ObjectProvider.getObject()` 를 `LogDemoController` , `LogDemoService` 에서 각각 한번씩 따로 호출해도 같은 HTTP 요청이면 같은 스프링 빈이 반환된다! ➡ 내가 직접 이걸 구분하려면 얼마나 힘들까 ㅠㅠ...

이 정도에서 끝내도 될 것 같지만... 개발자들의 코드 몇자를 더 줄이려는 욕심은 끝이 없다.

스코프와 프록시

이번에는 프록시 방식을 사용해보자.

```
@Component
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class MyLogger {
}
```

- 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.

이제 나머지 코드를 Provider 사용 이전으로 돌려두자.

```
package hello.core.web;

import hello.core.common.MyLogger;
import hello.core.logdemo.LogDemoService;
```

```java
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.servlet.http.HttpServletRequest;

@Controller
@RequiredArgsConstructor
public class LogDemoController {

    private final LogDemoService logDemoService;
    private final MyLogger myLogger;

    @RequestMapping("log-demo")
    @ResponseBody
    public String logDemo(HttpServletRequest request) {
        String requestURL = request.getRequestURL().toString();
        myLogger.setRequestURL(requestURL);

        myLogger.log("controller test");
        logDemoService.logic("testId");
        return "OK";
    }
}
```

```java
package hello.core.logdemo;

import hello.core.common.MyLogger;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class LogDemoService {
```

```
    private final MyLogger myLogger;

    public void logic(String id) {
        myLogger.log("service id = " + id);
    }

}
```

실행해보면 잘 동작하는 것을 확인할 수 있다.

코드를 잘 보면 `LogDemoController`, `LogDemoService`는 Provider 사용 전과 완전히 동일하다. 어떻게 된 것일까?

웹 스코프와 프록시 동작 원리

가짜를 넣고 실제기능을 호출할때 진짜를 찾아서 동작함.

먼저 주입된 myLogger를 확인해보자.

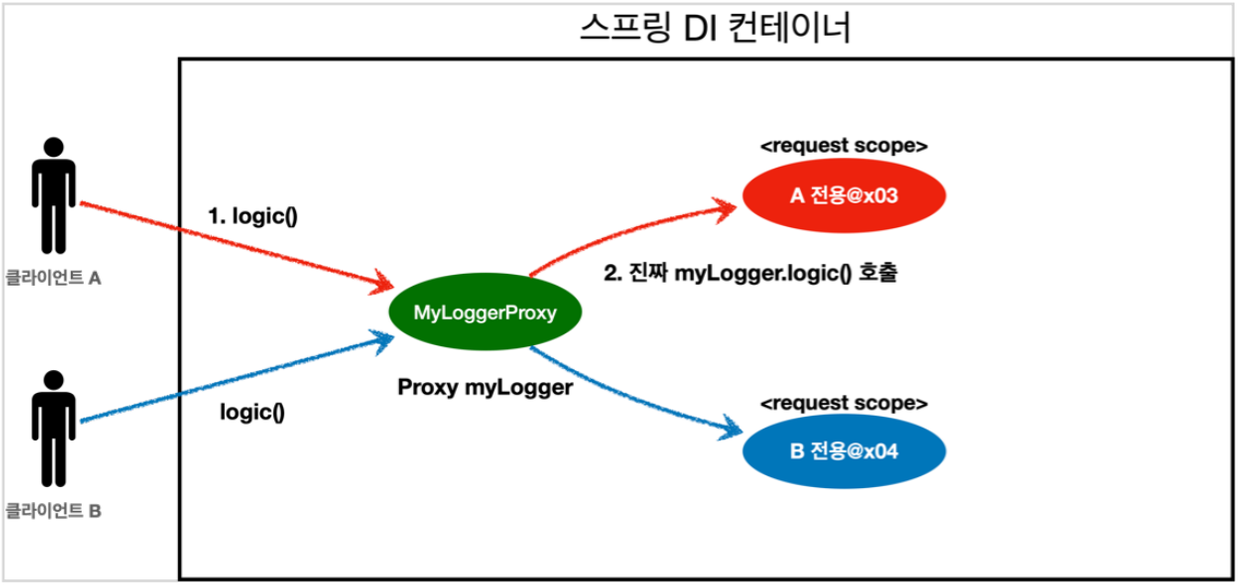
```
System.out.println("myLogger = " + myLogger.getClass());
```

출력결과

```
myLogger = class hello.core.common.MyLogger$$EnhancerBySpringCGLIB$$b68b726d
```

CGLIB라는 라이브러리로 내 클래스를 상속 받은 가짜 프록시 객체를 만들어서 주입한다.

- `@Scope`의 `proxyMode = ScopedProxyMode.TARGET_CLASS)`를 설정하면 스프링 컨테이너는 CGLIB라는 바이트코드를 조작하는 라이브러리를 사용해서, MyLogger를 상속받은 가짜 프록시 객체를 생성한다.
- 결과를 확인해보면 우리가 등록한 순수한 MyLogger 클래스가 아니라 `MyLogger$$EnhancerBySpringCGLIB`이라는 클래스로 만들어진 객체가 대신 등록된 것을 확인할 수 있다.
- 그리고 스프링 컨테이너에 "myLogger"라는 이름으로 진짜 대신에 이 가짜 프록시 객체를 등록한다.
- `ac.getBean("myLogger", MyLogger.class)`로 조회해도 프록시 객체가 조회되는 것을 확인할 수 있다.
- 그래서 의존관계 주입도 이 가짜 프록시 객체가 주입된다.



가짜 프록시 객체는 요청이 오면 그때 내부에서 진짜 빈을 요청하는 위임 로직이 들어있다.

- 가짜 프록시 객체는 내부에 진짜 myLogger를 찾는 방법을 알고 있다.
- 클라이언트가 `myLogger.logic()` 을 호출하면 사실은 가짜 프록시 객체의 메서드를 호출한 것이다.
- 가짜 프록시 객체는 request 스코프의 진짜 `myLogger.logic()` 를 호출한다.
- 가짜 프록시 객체는 원본 클래스를 상속 받아서 만들어졌기 때문에 이 객체를 사용하는 클라이언트 입장에서는 사실 원본인지 아닌지도 모르게, 동일하게 사용할 수 있다(다형성)

동작 정리

- CGLIB라는 라이브러리로 내 클래스를 상속 받은 가짜 프록시 객체를 만들어서 주입한다.
- 이 가짜 프록시 객체는 실제 요청이 오면 그때 내부에서 실제 빈을 요청하는 위임 로직이 들어있다.
- 가짜 프록시 객체는 실제 request scope와는 관계가 없다. 그냥 가짜이고, 내부에 단순한 위임 로직만 있고, 싱글톤 처럼 동작한다.

특징 정리

- 프록시 객체 덕분에 클라이언트는 마치 싱글톤 빈을 사용하듯이 편리하게 request scope를 사용할 수 있다.
- 사실 Provider를 사용하든, 프록시를 사용하든 핵심 아이디어는 진짜 객체 조회를 꼭 필요한 시점까지 지연처리 한다는 점이다.
- 단지 애노테이션 설정 변경만으로 원본 객체를 프록시 객체로 대체할 수 있다. 이것이 바로 다형성과 DI 컨테이너가 가진 큰 강점이다.
- 꼭 웹 스코프가 아니어도 프록시는 사용할 수 있다.

주의점

- 마치 싱글톤을 사용하는 것 같지만 다르게 동작하기 때문에 결국 주의해서 사용해야 한다.
- 이런 특별한 scope는 꼭 필요한 곳에만 최소화해서 사용하자, 무분별하게 사용하면 유지보수하기 어려워진다.