

# 10

## Chapter 10. 빈 스코프

### 빈 스코프란?

---

- 지금까지 우리는 스프링 빈이 스프링 컨테이너의 시작과 함께 생성돼서, 스프링 컨테이너가 종료될 때까지 유지된다고 배웠다.
- 이는 스프링 빈이 기본적으로 싱글톤 스코프로 생성되기 때문
- 스코프?

빈이 존재할 수 있는 범위를 의미

- 스프링이 지원하는 스코프 형태

- **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 스코프>

- **request**

웹 요청이 들어오고 나갈때까지 유지되는 스코프

- **session**

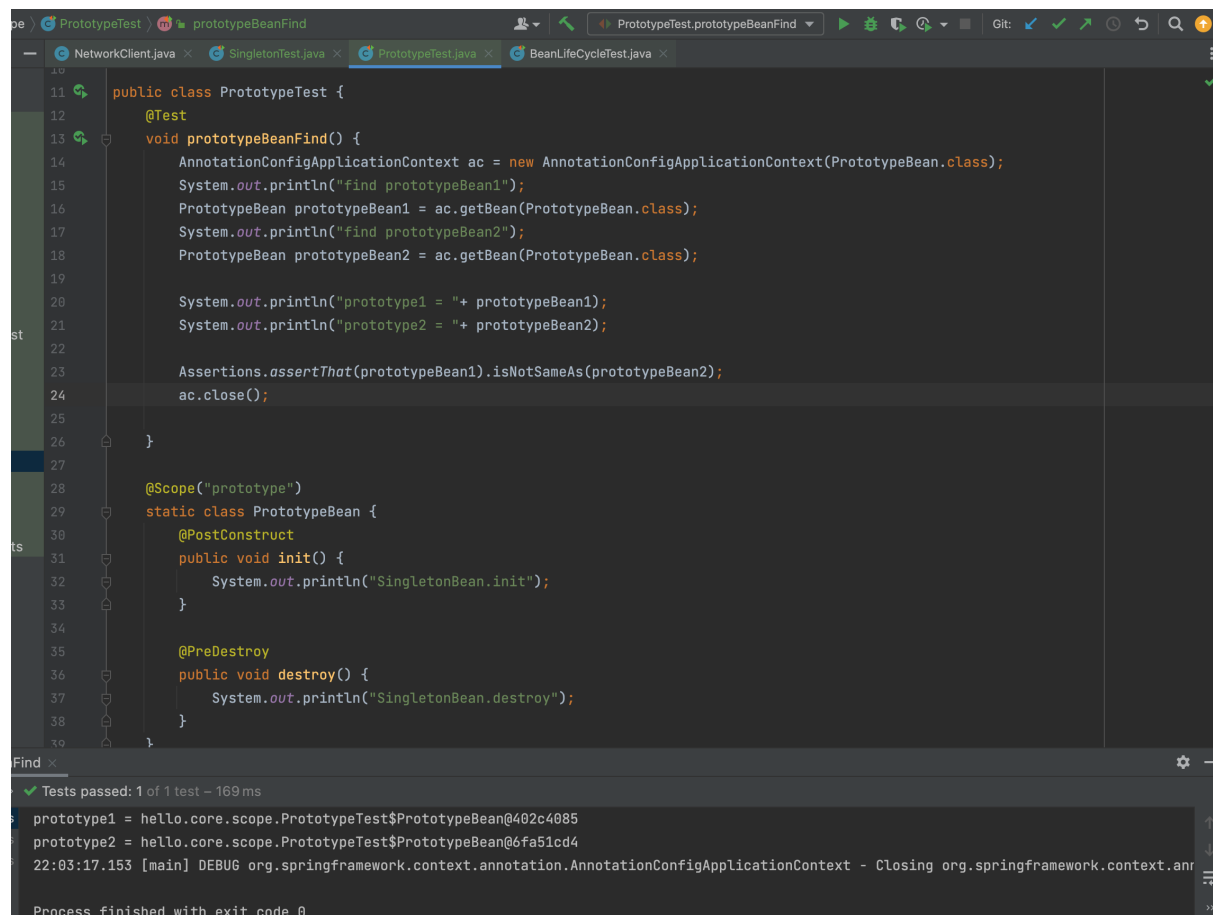
웹 세션이 생성되고 종료될 때까지 유지되는 스코프

- **application**

웹의 서블릿 컨텍스트와 같은 범위로 유지되는 스코프

## 프로토타입 스코프

1. 프로토타입 스코프 빈을 스프링 컨테이너에 요청한다.
2. 스프링 컨테이너는 이 시점에 프로토타입 빈을 생성하고, 필요한 의존관계를 주입한다.
3. 스프링 컨테이너는 생성한 프로토타입 빈을 클라이언트에 반환한다.
4. 반환 후 관리는 안함
5. 이후 스프링 컨테이너에 같은 요청이 오면 항상 새로운 프로토타입 빈을 생성해서 반환한다.



The screenshot shows an IDE with a Java file named `PrototypeTest.java`. The code defines a `PrototypeTest` class with a `@Test` method `prototypeBeanFind()`. This method creates an `AnnotationConfigApplicationContext` with `PrototypeBean.class`, retrieves two beans, and asserts they are not the same object. A static inner class `PrototypeBean` is also defined with `@PostConstruct` and `@PreDestroy` methods. The output window at the bottom shows the test passing and the two beans being different objects.

```
public class PrototypeTest {
    @Test
    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("prototype1 = " + prototypeBean1);
        System.out.println("prototype2 = " + prototypeBean2);

        Assertions.assertThat(prototypeBean1).isNotSameAs(prototypeBean2);
        ac.close();
    }

    @Scope("prototype")
    static class PrototypeBean {
        @PostConstruct
        public void init() {
            System.out.println("SingletonBean.init");
        }

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

Find x

Tests passed: 1 of 1 test - 169 ms

prototype1 = hello.core.scope.PrototypeTest\$PrototypeBean@402c4085  
prototype2 = hello.core.scope.PrototypeTest\$PrototypeBean@6fa51cd4  
22:03:17.153 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing org.springframework.context.annot

Process finished with exit code 0

- 핵심
  - 스프링 컨테이너에 요청할 때마다 새로 생성한다.
  - 스프링 컨테이너는 프로토타입 빈을 생성하고, 의존관계 주입, 초기화까지만 처리함

- 클라이언트에 빈을 반환하고, 이후 관리는 안한다. (3~4번)
- 프로토타입 빈을 관리해야하는 책임은 클라이언트 에 있다.
- 그래서 @PreDestroy 같은 종료 메서드가 호출되지 않는다.

## 프로토타입 스코프 - 싱글톤 빈과 함께 사용시 문제점

```

27 }
28
29 @Test
30 void singletonClientUserPrototype() {
31     AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(ClientBean.class, PrototypeBean.class);
32     ClientBean clientBean1 = ac.getBean(ClientBean.class);
33     int count1= clientBean1.logic();
34     assertThat(count1).isEqualTo(1);
35
36     ClientBean clientBean2 = ac.getBean(ClientBean.class);
37     int count2= clientBean2.logic();
38     assertThat(count2).isEqualTo(2);
39
40 }
41
42
43 @Scope("singleton")
44 static class ClientBean {
45     private final PrototypeBean prototypeBean; // 생성시점에 주입
46
47     @Autowired

```

- 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,  
처음 생성된 싱글톤 빈이 계속 유지되는 것이 문제
- 프로토타입 빈을 주입 시점에만 새로 생성하는 것이 아닌, 사용할 때마다 새로 생성해서 사용하고 싶은데?

→ 다음시간에 !!

## 프로토타입 스코프 - 싱글톤 빈과 함께 사용시 Provider로 문제 해결

- 그런데 싱글톤 빈은 생성 시점에만 의존관계 주입을 받기 때문에, 프로토타입 빈이 새로 생성되기는 하지만,

처음 생성된 싱글톤 빈이 계속 유지되는 것이 문제

- 프로토타입 빈을 주입 시점에만 새로 생성하는 것이 아닌, 사용할 때마다 새로 생성해서 사용하고 싶은데?

→ 싱글톤 빈이 프로토타입을 사용할 때마다 스프링 컨테이너에 새로 요청하는 것

- 지금 필요한 기능은 지정한 프로토타입 빈을 컨테이너에서 대신 찾아주는 딱 “DL” 정도의 기능만 제공하는 무언가 !

- DL : Dependency Lookup, 의존관계 조회(탐색)

- ObjectFactory, ObjectProvider

The screenshot shows an IDE with several tabs open. The active tab is `SingletonWithPrototypeTest1.java`, which contains the following code:

```

43
44 @Scope("singleton")
45 static class ClientBean {
46
47     @Autowired
48     private ObjectProvider<PrototypeBean> prototypeBeanProvider;
49
50     public int logic() {
51         PrototypeBean prototypeBean = prototypeBeanProvider.getObject();
52         prototypeBean.addCount();
53         int count = prototypeBean.getCount();
54         return count;
55     }
56 }
57

```

Below the code editor, the test results are displayed:

```

Test1.prototypeFind
>> Tests passed: 1 of 1 test - 171 ms

171 ms /Library/Java/JavaVirtualMachines/jdk-17.0.2.jdk/Contents/Home/bin/java ...
22:30:44.007 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext
22:30:44.014 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean
22:30:44.025 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean
22:30:44.026 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean
22:30:44.026 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean
22:30:44.027 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean
PrototypeBean.inithello.core.scope.SingletonWithPrototypeTest1$PrototypeBean@5524cca1
PrototypeBean.inithello.core.scope.SingletonWithPrototypeTest1$PrototypeBean@36804139

Process finished with exit code 0

```

- 핵심은 `ObjectProvider`가 지금 딱 필요한 DL 정도의 기능을 제공한다는 것

- 이 둘은 스프링에 의존함

- JSR-330 Provider

- 라이브러리 추가가 필요함
- `get()` 메소드 하나로 기능이 매우 단순
- 별도의 라이브러리가 필요함
- 자바 표준이므로 스프링 이외의 컨테이너에서도 사용 가능
- 정리
  - 프로토타입 빈을 언제 사용하는가? 실무에서는 거의 사용 안한다.

## 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.

- 웹 스코프
  - 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 관리됨
  - session
    - HTTP Session과 동일한 생명주기를 가지는 스코프

- application
  - 서블릿 컨텍스트와 동일한 생명주기를 가지는 스코프
- websocket
  - 웹 소켓과 동일한 생명주기를 가지는 스코프

## request 스코프 예제 만들기

---

- 웹 환경 추가
  - ▼ web 라이브러리 추가

```
plugins {
    id 'org.springframework.boot' version '2.6.6'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

// lombok 설정 추가 시작
configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}
// lombok 설정 추가 끝

repositories {
    mavenCentral()
}

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

    // lombok 라이브러리 추가 시작
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'

    testCompileOnly 'org.projectlombok:lombok'
```

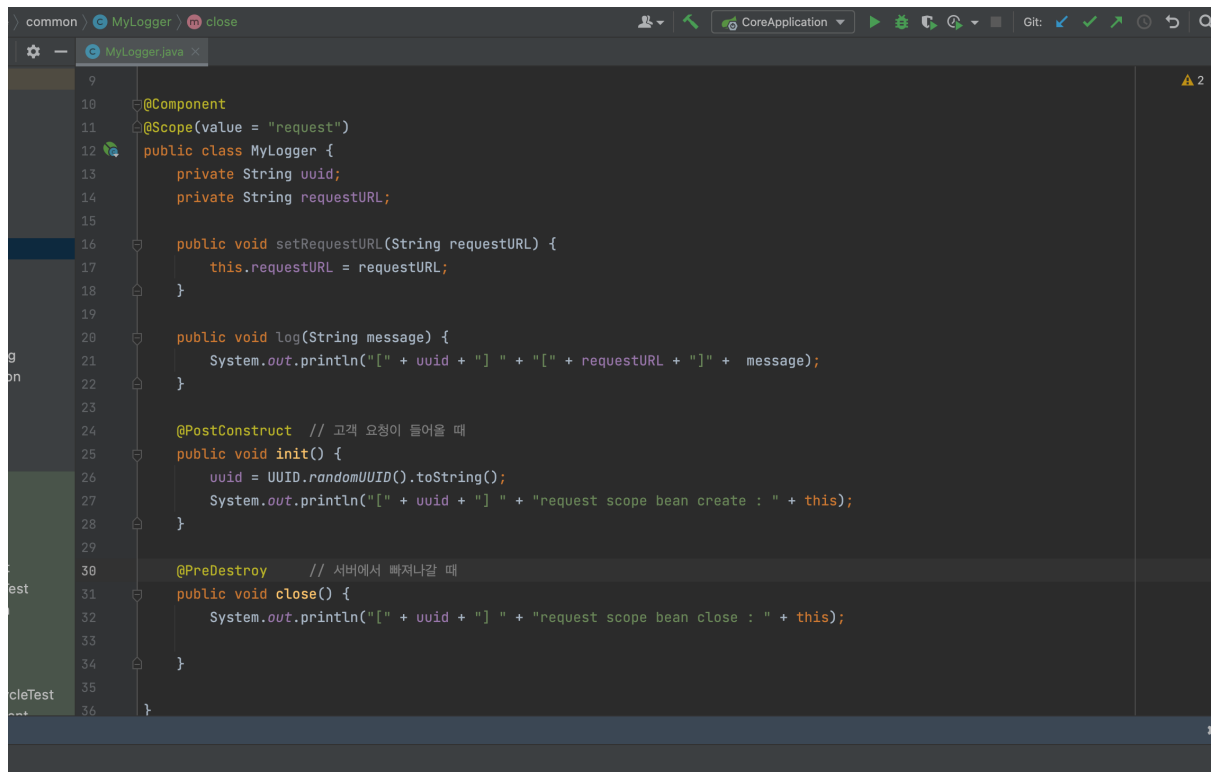
```

testAnnotationProcessor 'org.projectlombok:lombok'

testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
    useJUnitPlatform()
}

```



- 이제 이 빈은 HTTP 요청당 하나씩 생성됨
- UUID로 다른 요청과 구분함

## 스코프와 Provider

```

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 // 고객 요청이 들어올 때
    public void init() {
        uuid = UUID.randomUUID().toString();
        System.out.println "[" + uuid + " " + "request scope bean create : " + this);
    }

    @PreDestroy // 서버에서 빠져나갈 때
    public void close() {
        System.out.println "[" + uuid + " " + "request scope bean close : " + this);
    }
}

```

```

package hello.core.web;

import hello.core.common.MyLogger;
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.web;

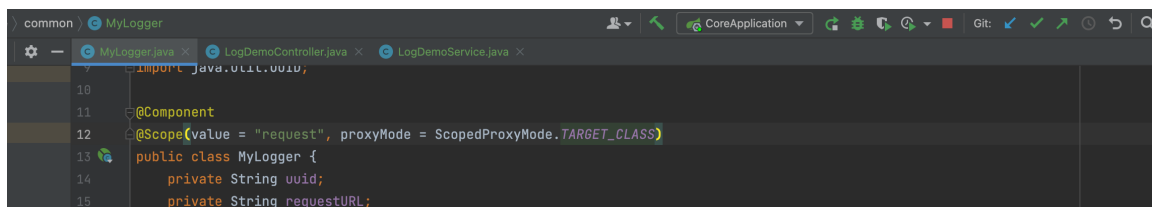
import hello.core.common.MyLogger;
import lombok.RequiredArgsConstructor;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class LogDemoService {
    private final ObjectProvider<MyLogger> myLoggerProvider;
    public void logic(String id) {
        MyLogger myLogger = myLoggerProvider.getObject();
        myLogger.log("service id = " + id );
    }
}

```

## 스코프와 프록시

- 프록시 모드를 추가함



- 이렇게 하면 MyLogger의 가짜 프록시 클래스를 만들어두고 HTTP request와 상관없이 가짜 프록시 클래스를 미리 주입해 둘 수 있다.
- 웹 스코프와 프록시 동작 원리

- CGLIB라는 라이브러리로 내 클래스를 상속받은 가짜 프록시 객체를 만들어서 주입한다,
- @Scope의 proxyMode ~~ 를 설정하면 스프링 컨테이너는 CGLIB라는 바이트 코드를 조작하는 라이브러리를 사용해서 MyLogger를 상속받은 가짜 프록시 객체를 생성한다.
- 결과를 확인해보면  

```
LogDemoController.logDemoclass
hello.core.common.MyLogger$$EnhancerBySpringCGLIB$$31ebcfff
```

 라는 클래스로 만들어진 객체가 대신 등록된 것을 확인할 수 있다.
- 의존 관계 주입도 이 가짜 프록시 객체가 주입된다.

#### • 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 들어온다.
- 가짜 프록시 객체는 실제 request scope과는 관계가 없다.
- 그냥 가짜이고, 내부에 단순한 위임 로직만 있음
- 싱글톤처럼 동작

#### • 특징 정리

- 프록시 객체 덕분에 클라이언트는 마치 싱글톤 빈을 사용하듯이 편리하게 request scope를 사용할 수 있다.
- 사실 Provider를 사용하든, 프록시를 사용하든 핵심은 “진짜 객체 조회를 꼭 필요한 시점까지 지연처리” 한다는 것

#### • 주의점

- 마치 싱글톤을 사용하는 것 같지만, 다르게 동작함
- 남발하면 유지보수 어려움