9️⃣

Section9. 빈 스코프

빈 스코프란?

스프링 빈이 스프링 컨테이너의 시작과 함께 생성되어서 스프링 컨테이너가 종료될 때 까지 유지된다고 학습했다.

이것은 스프링 빈이 기본적으로 싱글톤 스코프로 생성되기 때문이다.

스코프는 번역 그대로 빈이 존재할수 있는 범위를 뜻한다.

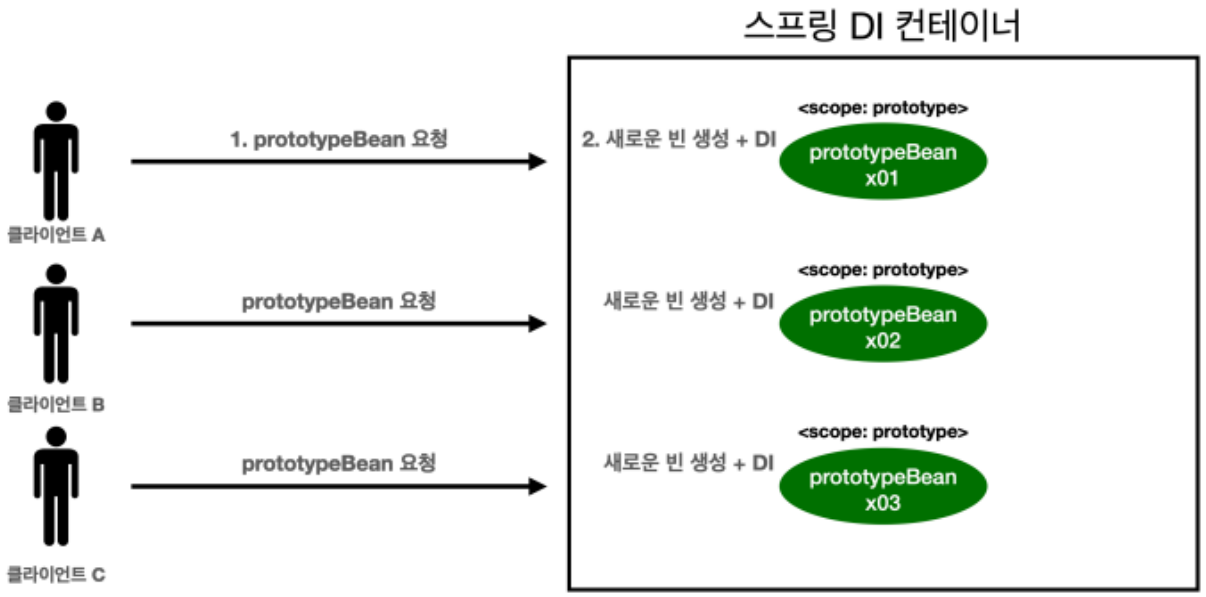
스프링은 다음과 같은 다양한 스코프를 지원한다.

- **싱글톤**: 기본 스코프, 스프링 컨테이너의 시작과 종료까지 유지되는 가장 넓은 범위 스코프
- **프로토타입**: 스프링 컨테이너는 프로토타입 빈의 생성과 의존관계 주입까지만 관여
  - 매우 짧은 범위의 스코프
- **웹 관련 스코프**
  - **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

프로토타입 스코프

프로토타입 스코프를 스프링 컨테이너에 조회하면 스프링 컨테이너는 **항상 새로운 인스턴스를 생성해서 반환**한다.

프로토타입 빈 요청1



- 프로토타입 스코프의 빈을 스프링 컨테이너에 요청
- 스프링 컨테이너는 이 시점에 프로토타입 빈을 생성, 필요한 의존관계 주입

프로토타입 빈 요청2

- 스프링 컨테이너는 생성한 프로토타입 빈을 클라이언트에 반환
- 이후 스프링 컨테이너에 같은 요청이 오면 항상 새로운 프로토타입 빈을 생성해서 반환

정리

스프링 컨테이너는 프로토타입 빈을 생성하고, 의존관계 주입, 초기화까지만 처리한다는 것이다.

프로토타입 **빈을 관리할 책임은 프로토타입 빈을 받은 클라이언트에 있다**. 그래서 @PreDestroy 같은 종료 메서드가 호출되지 않는다.

> 💡 실습 코드 작성

- 프로토타입 빈은 스프링 컨테이너에서 빈을 조회할 때 생성되고, 초기화 메서드도 실행된다.
- 완전히 다른 스프링 빈이 생성되고, 초기화도 2번 실행된 것을 확인할 수 있다.
- 프로토타입 빈은 스프링 컨테이너가 종료될 때 `@PreDestroy` 같은 종료 메서드가 전혀 실행되지 않는다.

프로토타입 빈의 특징 정리

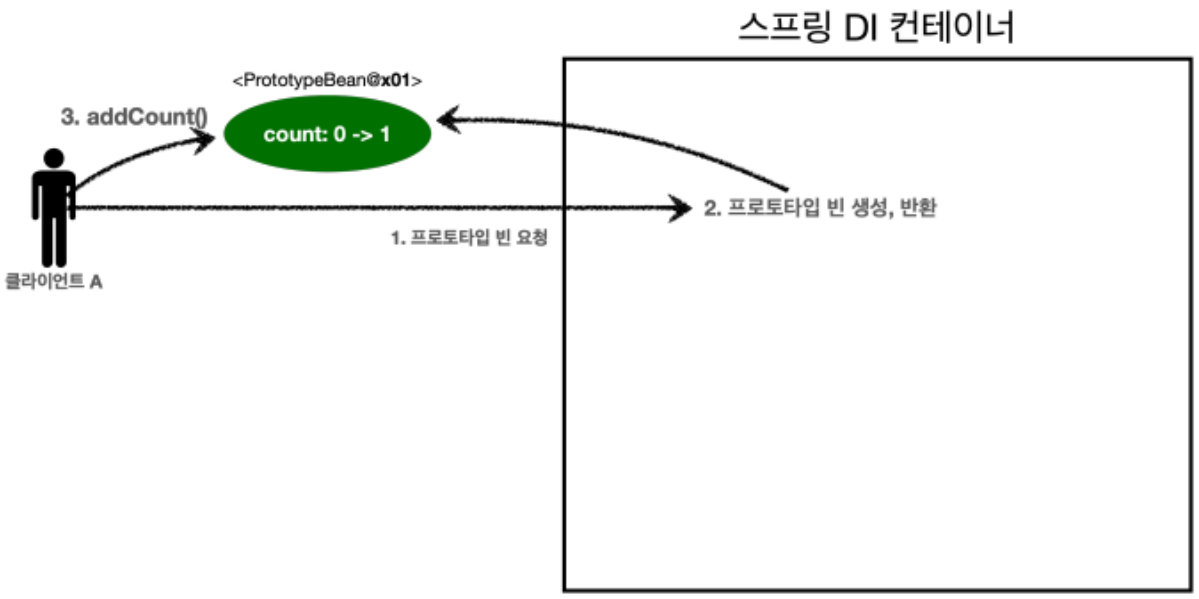
- 스프링 컨테이너에 요청할 때 마다 새로 생성된다.
- 스프링 컨테이너는 프로토타입 빈의 생성과 의존관계 주입 그리고 초기화까지만 관여한다.
- 종료 메서드가 호출되지 않는다. 클라이언트가 직접 해야한다.
- 프로토타입 빈은 프로토타입 빈을 조회한 클라이언트가 관리해야 한다.

프로토타입 스코프 - 싱글톤 빈과 함께 사용시 문제점

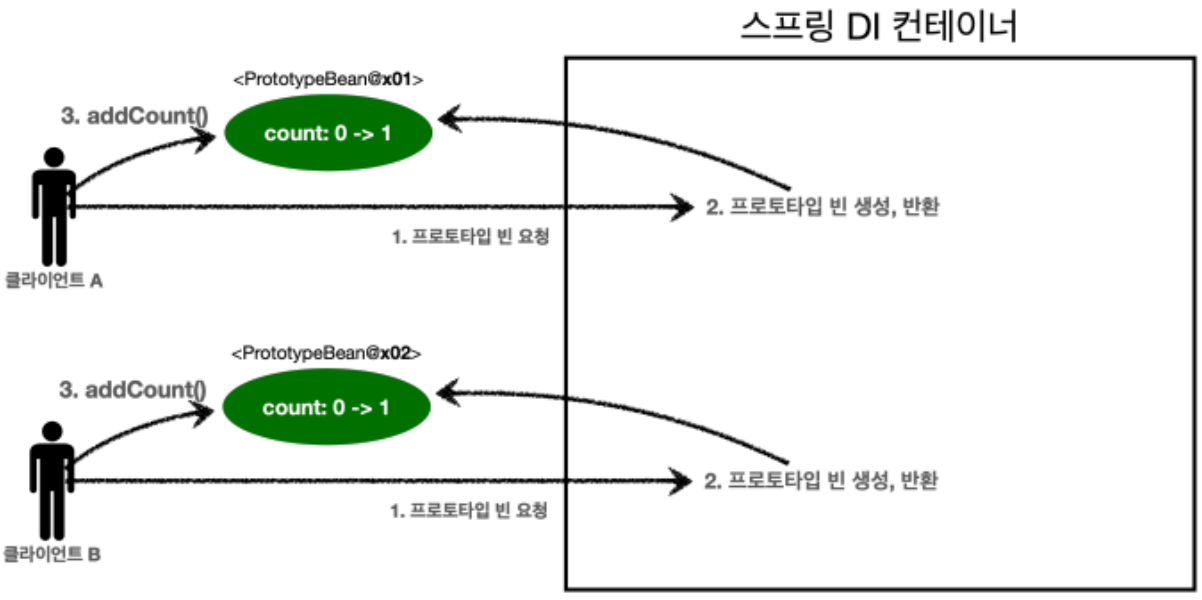
싱글톤 빈과 함께 사용할 때는 의도한 대로 잘 동작하지 않으므로 주의해야 한다.

스프링 컨테이너에 프로토타입 빈을 직접 요청하는 예제

프로토타입 빈 직접 요청



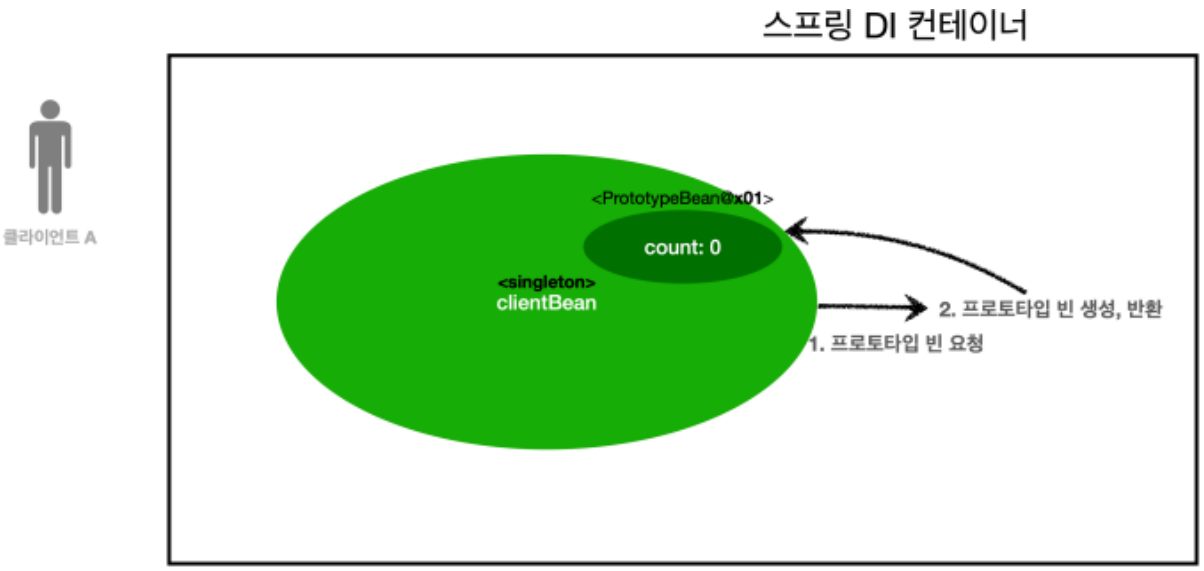
1. 클라이언트 A는 스프링 컨테이너에 프로토타입 빈 요청
2. 스프링 컨테이너는 프로토타입 빈을 새로 생성해서 반환(x01). 해당 빈 count 필드 값 0
3. 클라이언트는 조회한 프로토타입 빈에 `addCount()` 를 호출하면서 count + 1
4. 결과적으로 프로토타입 빈(x01)의 count = 1



1. 클라이언트B는 스프링 컨테이너에 프로토타입 빈을 요청한다.
2. 스프링 컨테이너는 프로토타입 빈을 새로 생성해서 반환(x02)한다. 해당 빈의 count 필드 값 0

3. 클라이언트는 조회한 프로토타입 빈에 addCount() 를 호출하면서 count 필드를 +1 한다.
4. 결과적으로 프로토타입 빈(x02)의 count는 1이 된다.

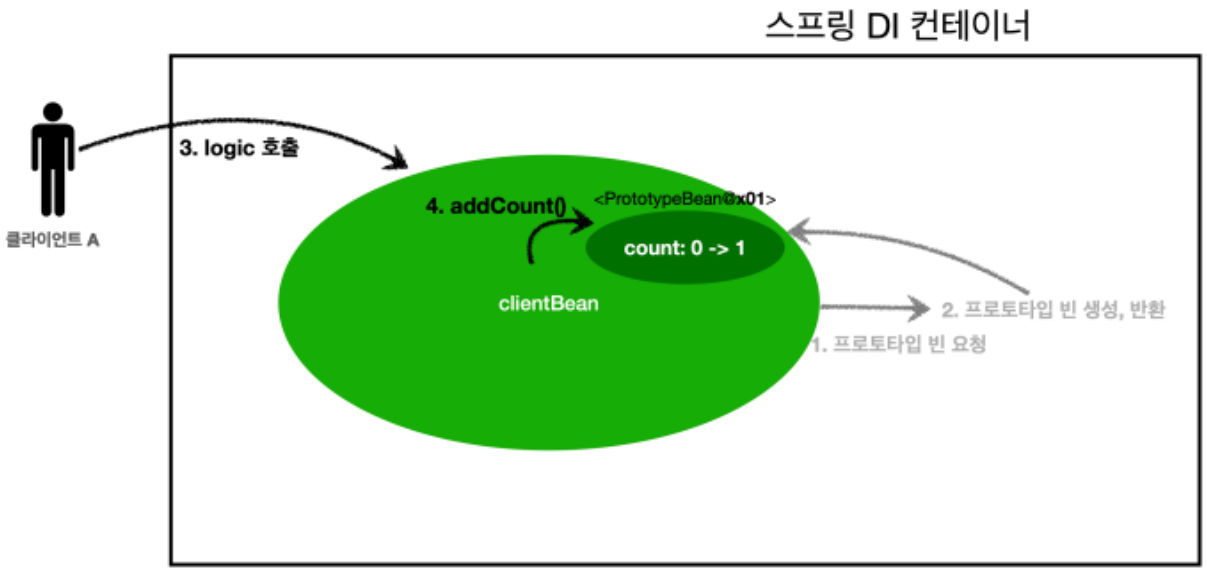
싱글톤 빈에서 프로토타입 빈 사용



- `clientBean` 은 싱글톤, 스프링 컨테이너 생성 시점에 생성되고, 의존관계 주입도 발생한다.

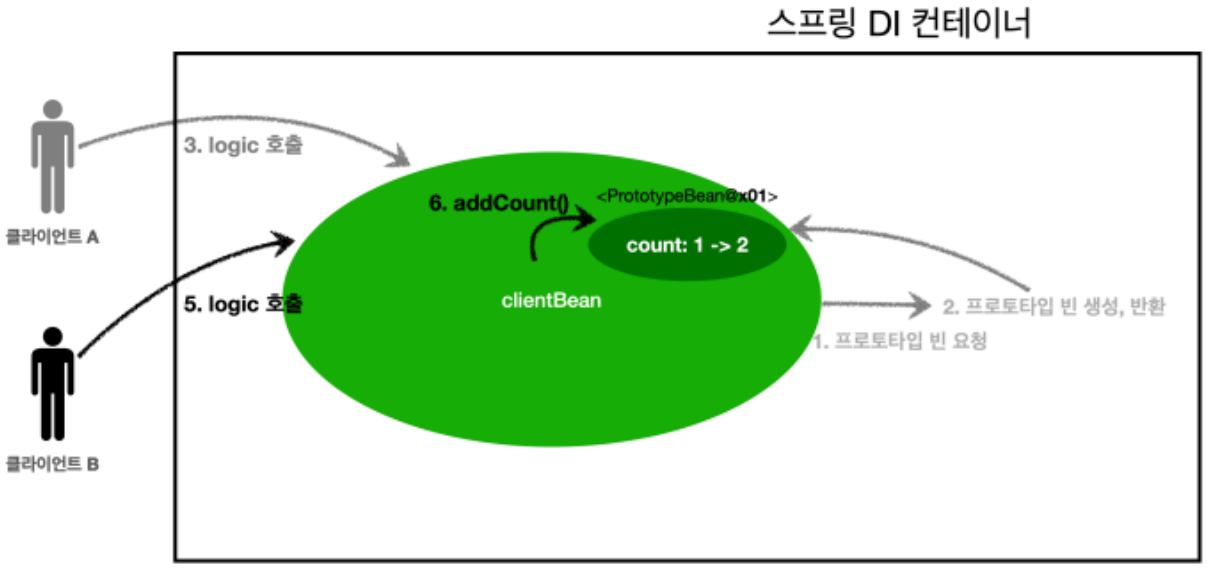
1. `clientBean`은 의존관계 자동 주입을 사용한다.
   주입 시점에 스프링 컨테이너에 프로토타입 빈을 요청한다.
2. 스프링 컨테이너는 프로토타입 빈을 생성해서 clientBean 에 반환한다.
   프로토타입 빈의 count 필드값 은 0이다.

- `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.



- 클라이언트 B는 `clientBean` 을 스프링 컨테이너에 요청해서 받는다. 싱글톤이므로 항상 같은 `clientBean` 이 반환된다.
- **`clientBean`이 내부에 가지고 있는 프로토타입 빈은 이미 과거에 주입이 끝난 빈이다.**
  주입 시점에 스프링 컨테이너에 요청해서 프로토타입 빈이 새로 생성이 된 것이지, 사용할 때마다 새로 생성되는 것이 아니다!

5. 클라이언트 B는 `clientBean.logic()` 을 호출한다.
6. `clientBean` 은 `prototypeBean`의 `addCount()` 를 호출해서 프로토타입 빈의 `count`를 증가한다. 원래 count 값이 1이었으므로 2가 된다.

> 💡 실습 코드 작성

싱글톤 빈은 생성 시점에만 의존관계 주입을 받기 때문에, **프로토타입 빈이 싱글톤 빈과 함께 계속 유지되는 것이 문제**다.

프로토타입 빈을 주입 시점에만 새로 생성하는게 아니라, 사용할 때 마다 새로 생성해서 사용하는 것을 원할 것이다.

> 💡 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`

프로토타입 스코프 - 싱글톤 빈 함께 사용 시 Provider로 문제 해결

ObjectFactory, ObjectProvider

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

```
@Autowired
private ObjectProvider<PrototypeBean> prototypeBeanProvider;
public int logic() {
     PrototypeBean prototypeBean = prototypeBeanProvider.getO
     prototypeBean.addCount();
```

```
        int count = prototypeBean.getCount();
        return count;
}
```

- `ObjectProvider` 의 `getObject()` 를 호출하면 내부에서는 스프링 컨테이너를 통해 해당 빈을 찾아서 반환한다. **(DL)**
- `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

스프링 부트 3.0은 `jakarta.inject.Provider` 를 사용한다.

```
// 다음 라이브러리를 gradle에 추가
// JSR-330 Provider 라이브러리 추가
implementation 'jakarta.inject:jakarta.inject-api:2.0.1'

@Autowired
private Provider<PrototypeBean> provider;
public int logic() {
      PrototypeBean prototypeBean = provider.get();
      prototypeBean.addCount();
      int count = prototypeBean.getCount();
      return count;
}
```

- `provider` 의 `get()` 을 호출, 내부에서 스프링 컨테이너를 통해 해당 빈을 찾아서 반환한다. (DL)
- `get()` 메서드 하나로 기능이 매우 단순하다.
- 별도의 라이브러리가 필요하다.
- 자바 표준이므로 스프링이 아닌 **다른 컨테이너에서도 사용**할 수 있다.

정리

실무에서 웹 애플리케이션을 개발해보면, 싱글톤 빈으로 대부분의 문제를 해결할 수 있기 때문에
프로토타입 빈을 직접적으로 사용하는 일은 매우 드물다.

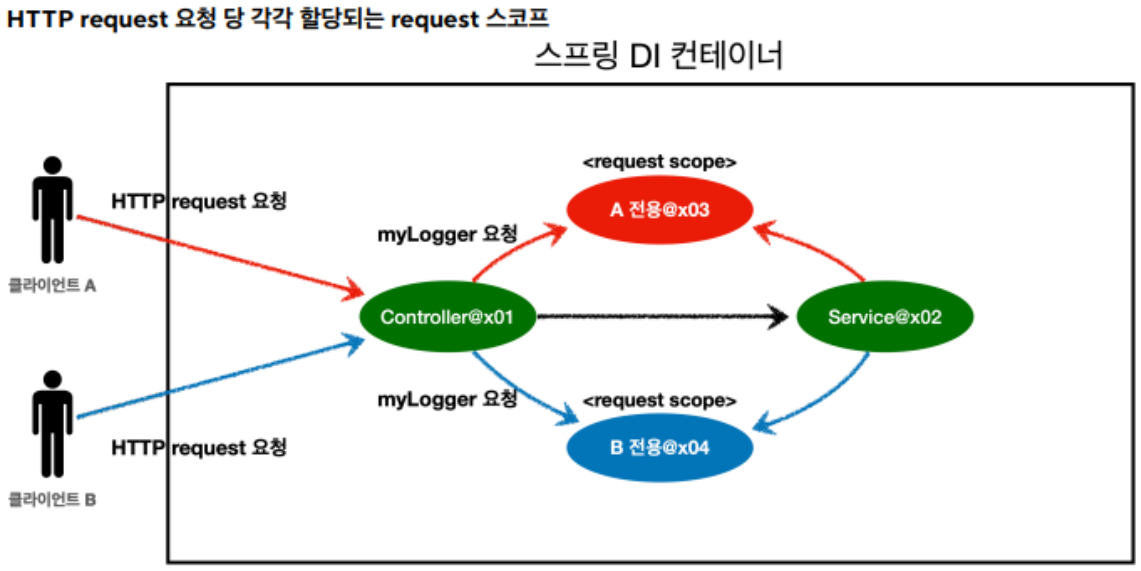
웹 스코프

웹 스코프는 웹 환경에서만 동작한다.

웹 스코프는 프로토타입과 다르게 스프링이 해당 스코프의 종료시점까지 관리한다.

종류

- `request`: **HTTP 요청 하나**가 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



---

request 스코프 예제 만들기

웹 스코프는 웹 환경에서만 동작하므로 web 환경이 동작하도록 라이브러리 추가

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

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

> 💡 실습 코드 작성

- `@Scope(value = "request")` 를 사용해서 request 스코프로 지정했다. 이제 이 빈은 HTTP 요청 당 하나씩 생성되고, HTTP 요청이 끝나는 시점에 소멸된다.
- `requestURL` 은 이 빈이 생성되는 시점에는 알 수 없으므로, 외부에서 `setter`로 입력 받는다.

> 💡 `requestURL`을 `MyLogger`에 저장하는 부분은 컨트롤러 보다는 공통 처리가 가능한 **스프링 인터셉터나 서블릿 필터 같은 곳을 활용**하는 것이 좋다.

실제는 기대와 다르게 애플리케이션 실행 시점에 오류 발생

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

스코프와 Provider

첫번째 해결방안은 앞서 배운 `Provider`를 사용하는 것이다.

```
// LogDemoController
@Controller
@RequiredArgsConstructor
public class LogDemoController {

    private final LogDemoService logDemoService;
    // private final MyLogger myLogger;
    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";
    }
}

// LogDemoService
@Service
@RequiredArgsConstructor
public class LogDemoService {

    // private final MyLogger myLogger;
    private final ObjectProvider<MyLogger> myLoggerProvider;

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

- `ObjectProvider` 덕분에 `ObjectProvider.getObject()` 를 호출하는 시점까지 `request scope` 빈의 생성을 지연할 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!

이제 코드를 더 줄여보자.

스코프와 프록시

```
// MyLogger

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

// 나머지 코드를 Provider 사용 이전으로 돌려두자
```

- `proxyMode = ScopedProxyMode.TARGET_CLASS` 를 추가해주자
  - 적용 대상이 인터페이스가 아닌 클래스면 `TARGET_CLASS` 를 선택
  - 적용 대상이 인터페이스면 `INTERFACES` 를 선택
- `MyLogger`의 가짜 프록시 클래스를 만들어두고 HTTP request와 상관 없이 가짜 프록시 클래스를
  다른 빈에 미리 주입해 둘 수 있다.

웹 스코프와 프록시 동작 원리

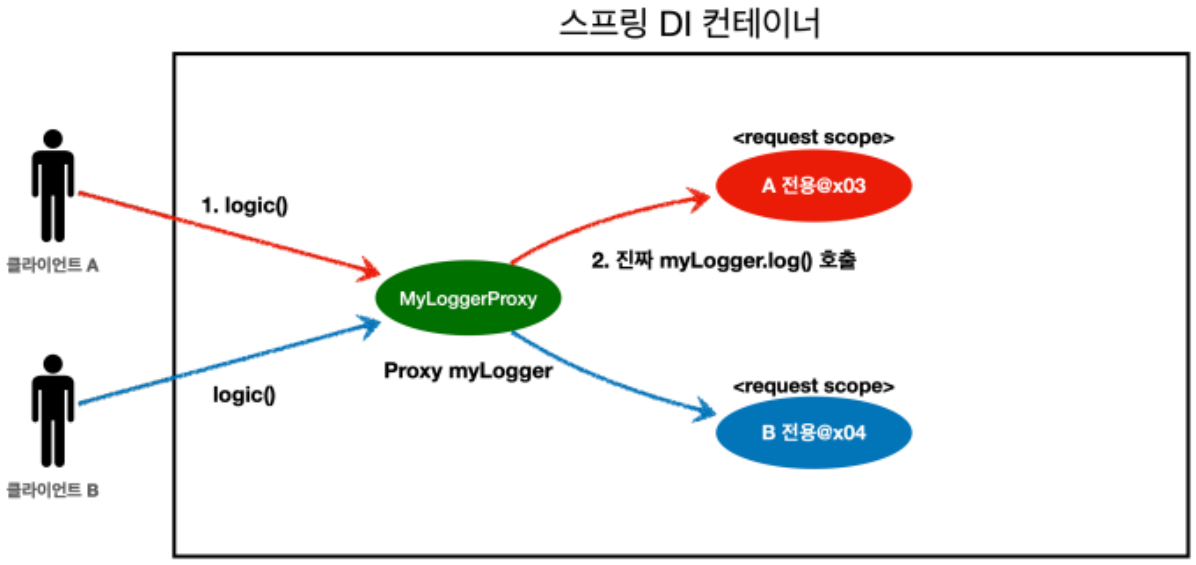
주입된 myLogger 확인

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

// 출력결과
// myLogger = class hello.core.common.MyLogger$$EnhancerBySpr
```

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

- 스프링 컨테이너는 `CGLIB` 라는 바이트코드를 조작하는 라이브러리를 사용해서, `MyLogger`를 상속받은 가짜 프록시 객체를 생성한다.
- 결과를 확인해보면 우리가 등록한 순수한 `MyLogger` 클래스가 아니라 `MyLogger$$EnhancerBySpringCGLIB` 클래스로 만들어진 객체가 등록된 것을 확인할 수 있다.
- 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.

- 클라이언트가 myLogger.log() 을 호출하면 사실은 가짜 프록시 객체의 메서드를 호출한 것이다.
- 가짜 프록시 객체는 request 스코프의 진짜 myLogger.log() 를 호출한다.

동작 정리

1. CGLIB라는 라이브러리로 내 클래스를 상속 받은 가짜 프록시 객체를 만들어서 주입한다.
2. 가짜 프록시 객체는 실제 요청이 오면 그때 내부에서 실제 빈을 요청하는 위임 로직이 들어있다.

3. 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.