스프링 웹 스코프를 활용하여 빈 생성하기

미션	[백엔드] 스프링 핵심 원리 - 기본
상태	진행 중
설명	Spring 프레임워크의 웹 스코프(request, session, application 등)를 활용하여 빈을 생성합니다. 웹 스코프 빈 생성 과정을 코드로 구현하고, 실행 결과를 확인합니다. 웹 스코프 빈 생성 과정과 실행 결과 스크린샷을 결과물로 제출합니다.

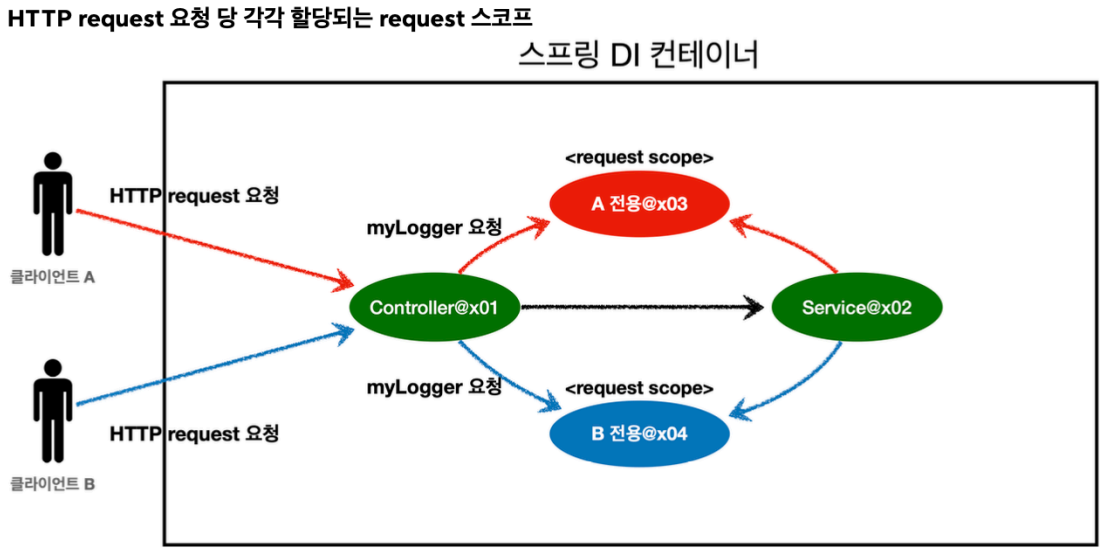
웹 스코프

특징

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

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 소캣과 동일한 생명주기를 가지는 스코프



Request 스코프 예제

동시에 여러 HTTP 요청이 오면 정확이 어떤 요청이 남긴 로그인지 구분하기 어려운데, 이때 사용하기 좋은 것이 request 스코프다.

MyLogger.java

```
package hello.core.common;

import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

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 + "] " + messag
e);
    }

    @PostConstruct
    public void init() {
        uuid = UUID.randomUUID().toString();
        System.out.println("[" + uuid + "] request scope bean create:" + this);
    }
    @PreDestroy
    public void close() {
        System.out.println("[" + uuid + "] request scope bean close:" + this);
    }
}
```

- 로그를 출력하기 위한 `MyLogger` 클래스
- `@Scope(value="request")`를 사용해서 request 스코프로 지정. 이제 이 빈은 HTTP 요청 당 하나씩 생성되고, HTTP 요청이 끝나는 시점에 소멸된다.

LogDemoController.java

```
package hello.core.web;

import hello.core.common.MyLogger;
import hello.core.logdemo.LogDemoService;
import jakarta.servlet.http.HttpServletRequest;
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;

@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";
    }
}
```

- Logger 동작 테스트 코드

LogDemoService.java

```
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 MyLogger myLogger;

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

```
    }
}
```

- 서비스 계층에서도 로그 출력
- request scope를 사용하지 않고 파라미터로 모든 정보를 서비스 계층으로 넘긴다면 파라미터가 많아져 지저분해질 것이다. 추가로 웹과 관련된 정보(requestURL 등)가 웹과 관련없는 서비스 계층까지 넘어가게 된다.
    - 웹 관련 코드는 컨트롤러에서만 사용해야 한다. 서비스 계층은 웹 기술에 종속되지 않고, 순수하게 유지하는 것이 유지보수 관점에서 좋다.

여기까지 작성하면 실행 시 오류가 발생한다. 스프링 애플리케이션을 실행하는 시점에 싱글톤 빈은 생성해서 주입 가능하지만, request 스코프 빈은 아직 생성되지 않는다. 이 빈은 실제 클라이언트의 요청이 있어야 생성할 수 있다.

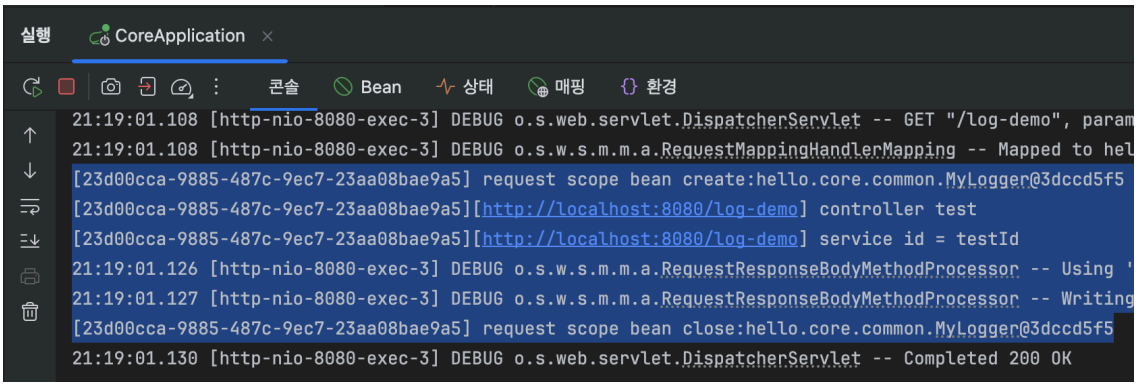
스코프와 Provider

앞선 문제를 해결하기 위한 방안으로 Provider를 사용하는 것이다. 간단히 `ObjectProvider`를 사용해보자.

```
@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";
```

```
    }
}
```

```
@Service
@RequiredArgsConstructor
public class LogDemoService {

    private final ObjectProvider<MyLogger> myLoggerProvider;

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



웹 브라우저에서 http://localhost:8080/log-demo 입력 시 출력되는 로그

- `ObjectProvider` 덕분에 `ObjectProvider.getObject()` 를 호출하는 시점까지 request scope 빈의 생성을 지연할 수 있다.
- `ObjectProvider.getObject()` 를 호출하는 시점에는 HTTP 요청이 진행중이므로 request scope 빈의 생성이 정상 처리된다.
- `ObjectProvider.getObject()` 를 `LogDemoController` , `LogDemoService` 에서 각각 호출해도 같은 HTTP 요청이면 같은 스프링이 반환된다.

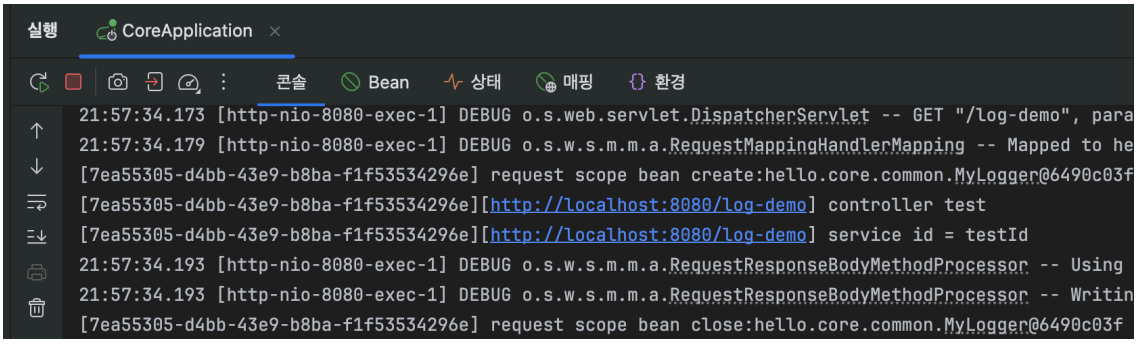
스코프와 프록시

```
@Component
@Scope(value="request", proxyMode = ScopedProxyMode.TARGET_CLAS
```

```
S)
public class MyLogger { }
```

- `proxyMode = ScopedProxyMode.TARGET_CLASS)` 추가
  - 적용 대상이 클래스면 TARGET_CLASS 선택
  - 적용 대상이 인터페이스면 INTERFACES 선택

코드 실행 전, Provider 적용 전으로 돌려주자.

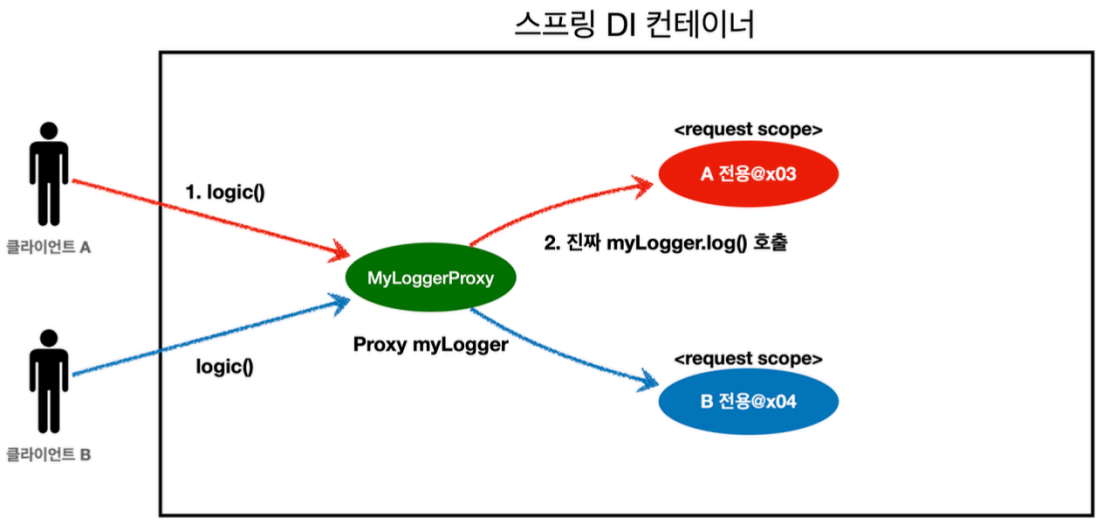


역시나 잘 동작한다.

웹 스코프와 프록시 동작 원리

> CGLIB라는 라이브러리로 내 클래스를 상속 받은 가짜 프록시 객체를 만든다.

- `@Scope`의 `proxyMode = ScopedProxyMode.TARGET_CLASS)`를 설정하면 스프링 컨테이너는 CGLIB라는 바이트코드를 조작하는 라이브러리를 사용하여 `MyLogger`를 상속받은 가짜 프록시 객체를 생성한다.
- 그리고 스프링 컨테이너에 “myLogger”라는 이름으로 진짜 대신에 이 가짜 프록시 객체를 등록한다.
- 의존관계 주입도 이 가짜 프록시 객체가 주입된다.



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

- 가짜 프록시 객체는 내부에 진짜 `myLogger` 를 찾는 방법을 알고 있다.
- 클라이언트가 `myLogger.log()`를 호출하면 사실은 가짜 프록시 객체의 메서드를 호출한 것이다.
- 가짜 프록시 객체는 request 스코프의 진짜 `myLogger.log()`를 호출한다.
- 가짜 프록시 객체는 원본 클래스를 상속 받아서 만들어졌기에 클라이언트는 동일하게 사용할 수 있다. (다형성)