

Spring week 6

유채린

chaerin.du.ub@gmail.com



해당 내용은 인프런 강의를 정리한 내용입니다.
잘못된 내용에 대하여 피드백주시면 감사하겠습니다!

(인프런 아이콘 클릭 시, 참고한 인프런 강의 사이트로 이동합니다.)

Index

- 빈 생명주기 콜백
- 빈 스코프

빈 생명주기 콜백

스프링 빈 라이프사이클 (싱글톤)

스프링 컨테이너 생성

New ApplicationContext

스프링 빈 생성

- ✓ @ComponentScan
- ✓ 빈 설정 파일 @Configuration

의존관계 주입

- ✓ 생성자 주입
- ✓ 수정자(setter) 주입
- ✓ 필드 주입

초기화 콜백

빈 사용

소멸 전 콜백

스프링 종료

- ✓ 인터페이스 (InitializingBean)
- ✓ 설정 정보에 초기화 메서드 지정 (커스텀)
- ✓ @PreDestroy 애노테이션

- ✓ 인터페이스 (DisposableBean)
- ✓ 설정 정보에 종료 메서드 지정 (커스텀)
- ✓ @PreDestroy 애노테이션

ConfigurableApplicationContext.close()

객체의 생성과 초기화는 분리하자

메모리 할당

동작

- 생성자는 필수 정보(파라미터)를 받고, 메모리를 할당해서 객체를 생성하는 책임을 가진다.
- 반면에 초기화는 이렇게 생성된 값들을 활용해서 외부 커넥션을 연결하는 등 무거운 동작을 수행한다.
- 생성자 안에 무거운 초기화 작업을 함께 하기 보다는 객체 생성과 초기화를 분리하여 관리하는 게 유지 보수 관점에서 좋다.
- 물론 초기화 작업이 단순한 경우에는 생성자에서 한 번에 다 처리하는 게 더 나을 수도 있다.

인터페이스 사용

- 초기화는 InitializingBean, 소멸은 DisposableBean 인터페이스를 상속받아 사용 가능하다.

```
import org.springframework.beans.factory.DisposableBean;  
import org.springframework.beans.factory.InitializingBean;  
  
public class NetworkClient implements InitializingBean, DisposableBean {  
    private String url;
```

```
@Override  
public void afterPropertiesSet() throws Exception {  
}  
  
@Override  
public void destroy() throws Exception {  
}
```

- 단점
 - 스프링 전용 인터페이스로 스프링 전용 인터페이스에 의존한다.
 - 초기화, 소멸 메서드의 이름 변경 불가능
 - 내가 코드를 고칠 수 없는 외부 라이브러리에 적용 불가능
- 해당 방식은 스프링 초창기에 나온 방법으로 거의 사용하지 않는다.

빈 등록 초기화, 소멸 메서드 지정

- 설정 정보에 초기화 메서드, 소멸 메서드 지정한다.
 - @Bean (initMethod = "init", destroyMethod = "close")
- 설정 정보 사용 특징
 1. 메서드 이름 자유롭게 가능
 2. 스프링 빈이 스프링 코드에 의존하지 않는다.
 3. 설정 정보를 사용함으로써 코드를 고칠 수 없는 외부 라이브러리에도 초기화 메서드와 종료 메서드 적용이 가능하다.
- @Bean destroyMethod 속성의 특별한 기능
 - 기본값이 (inferred) 추론 으로 등록되어 있어 close, shutdown이라는 이름의 메서드를 자동으로 호출해준다.
-> 종료 메서드를 추론해서 호출
 - 직접 스프링 빈으로 등록하면 종료 메서드는 따로 적어주지 않아도 잘 동작한다.
 - 추론기능 사용하기 싫은 경우 destroyMethod="" 처럼 빈 공백을 지정하면 된다.

```
@Configuration
static class LifecycleConfig {
    @Bean (initMethod = "init", destroyMethod = "close")
    public NetworkClient networkClient() {
        NetworkClient networkClient = new NetworkClient();
        networkClient.setUrl("http://hello-spring.dev");
        return networkClient;
    }
}
```


애노테이션 사용

- 초기화는 @PostConstruct, 소멸은 @PreDestroy 애노테이션을 이용한다.

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
// JSR-250 자바 표준
public class NetworkClient {
```

```
@PostConstruct
public void init() throws Exception {
    System.out.println("NetworkClient.init");
    connect();
    call("초기화 연결 메시지");
}

@PreDestroy
public void close() throws Exception {
    System.out.println("NetworkClient.close");
    disconnect();
}
```

- 최신 스프링에서 가장 권장하는 방법
- 매우 편리하다.
- 스프링이 아닌 다른 컨테이너에서도 동작한다.
- 컴포넌트 스캔과 잘 어울린다.
- 유일한 단점은 외부 라이브러리에서는 적용하지 못한다.
외부 라이브러리의 초기화, 종료는 @Bean의 기능을 사용하자 @Bean (initMethod = "init", destroyMethod = "close")

빈 스코프

빈 스코프

- 빈이 존재할 수 있는 범위

- 다양한 스코프를 지원하는 스프링

- ✓ 싱글톤

기본 스코프(default)로 스프링 컨테이너의 시작과 종료까지 유지되는 가장 넓은 범위의 스코프

- ✓ 프로토타입

스프링 컨테이너는 프로토타입의 빈 생성과 의존관계 주입까지만 관여하고 더는 관리하지 않는 매우 짧은 범위의 스코프

- ✓ 웹 관련 스코프

1. request: 웹 요청(HTTP 요청)이 들어오고 나갈 때까지 유지되는 스코프
2. session: 웹 세션(HTTP Session)이 생성되고 종료될 때까지 유지되는 스코프 ex) 로그인
3. application: 웹의 서블릿 컨텍스트(ServletContext)와 같은 범위로 유지되는 스코프
4. websocket: 웹 소켓과 동일한 생명주기를 갖는 스코프

빈 스코프 지정하는 방법

1. 컴포넌트 스캔을 통한 자동 등록

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

2. 수동 등록

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

프로토타입 스코프

- 항상 같은 인스턴스의 스프링 빈을 반환 받는 싱글톤 스코프 빈과 다르게 프로토타입 스코프를 조회하면 스프링 컨테이너는 항상 새로운 인스턴스를 생성하고 필요한 의존 관계를 주입한 후 클라이언트에게 반환한다.
- 핵심은 스프링 컨테이너는 프로토타입 빈을 생성하고 의존관계 주입 후 초기화 까지만 처리한다는 것이다. 클라이언트에게 빈을 반환한 후, 스프링 컨테이너는 생성된 프로토타입 빈을 관리하지 않는다. 종료 메서드가 호출되지 않는다.
- 프로토타입 빈을 관리하는 책임은 프로토타입 빈을 받은 클라이언트에게 있어서 종료 메서드에 대한 호출도 클라이언트가 직접 해야 한다.
- 싱글톤 빈은 스프링 컨테이너 생성 시점에 초기화 메서드가 실행 되지만, 프로토타입 스코프의 빈은 스프링 컨테이너에서 빈을 조회할 때 생성되고, 초기화 메서드도 실행된다.

프로토타입 스코프

- 싱글톤 빈과 함께 사용시 문제점

- 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.
- 우리가 원하는 건 사용할 때마다 스프링 컨테이너를 호출하여 새로 생성되는 것!
- 어떻게 하면 사용할 때 마다 항상 새로운 프로토타입 빈을 생성할 수 있을까?
→ Provider

interface

ObjectFactory, ObjectProvider

- 지정한 빈을 컨테이너에서 대신 찾아주는 DL 서비스를 제공하는 것이 바로 ObjectProvider
 - Dependency Lookup(DL) 의존관계: 직접 필요한 의존 관계를 찾는 것
- ObjectFactory는 과거 버전, ObjectProvider는 ObjectFactory에서 편의 기능 추가되어 만들어짐

```
static class ClientBean {  
  
    @Autowired  
    private ObjectProvider<PrototypeBean> prototypeBeanProvider;  
  
    public int logic() {  
        PrototypeBean prototypeBean = prototypeBeanProvider.getObject();  
        prototypeBean.addCount();  
        int count = prototypeBean.getCount();  
        return count;  
    }  
}
```

항상 새로운 프로토타입 빈 생성

DL
내부에서는 스프링 컨테이너를
통해 해당 빈을 찾아서 반환한다.

스프링이 제공하는 기능을 사용하지만,
기능이 단순하여 단위테스트 만들거나 mock코드 만들기 훨씬 쉬워진다.

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

Spring에 의존하지 않는 방식. 자바 표준

JSR-330 Provider

- javax.inject.Provider
- build.gradle에 javax.inject:javax.inject:1 라이브러리 추가

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter' // spring boot starter  
    //web 라이브러리 추가  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    // JSR-330 Provider  
    implementation 'javax.inject:javax.inject:1'  
    //lombok 라이브러리 추가 시작  
}
```

build.gradle

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

항상 새로운 프로토타입 빈 생성
DL
내부에서는 스프링 컨테이너를
통해 해당 빈을 찾아서 반환한다.

스프링이 제공하는 기능을 사용하지만,
기능이 단순하여 단위테스트 만들거나 mock코드 만들기 훨씬 쉬워진다.

- 특징
 - get() 메서드 하나로 기능이 매우 단순하다.
 - 별도의 라이브러리 필요하다.
 - 자바 표준으로 스프링이 아닌 다른 컨테이너에서도 사용 가능하다.

Provider 정리

- 매번 사용할 때마다 의존관계 주입이 완료된 새로운 객체가 필요하면 사용한다.
- 실무에서 웹 애플리케이션 개발하다 보면 싱글톤 빈으로 대부분의 문제를 해결할 수 있기 때문에 프로토타입 빈을 직접적으로 사용하는 일은 드물다.
- ObjectProvider, JSR-330 Provider 등은 프로토타입 뿐만 아니라 DL이 필요한 경우는 언제든지 사용 가능하다.

```
/**
 * Provides instances of {@code T}. Typically implemented by an injector. For
 * any type {@code T} that can be injected, you can also inject
 * {@code Provider<T>}. Compared to injecting {@code T} directly, injecting
 * {@code Provider<T>} enables:
 *
 * <ul>
 * <li>retrieving multiple instances.</li>
 * <li>lazy or optional retrieval of an instance.</li>
 * <li>breaking circular dependencies.</li>
 * <li>abstracting scope so you can look up an instance in a smaller scope
 *     from an instance in a containing scope.</li>
 * </ul>
 *
 * <p>For example:
 *
 * <pre>
 * class Car {
 *     @Inject Car(Provider<Seat> seatProvider) {
 *         Seat driver = seatProvider.get();
 *         Seat passenger = seatProvider.get();
 *         ...
 *     }
 * }
 * </pre>
 */
```

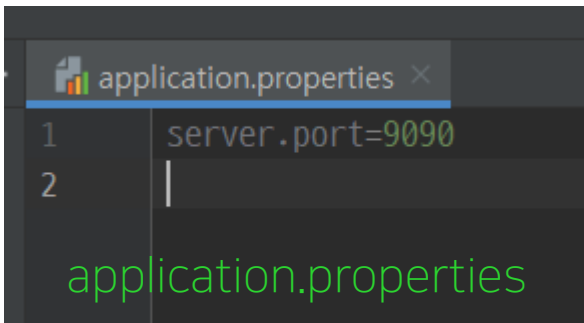
Provider.java 를 참고하자.

웹 스코프

- 웹 환경에서만 동작한다.
- 프로토타입과 다르게 스프링이 해당 스코프의 종료시점까지 관리한다. → 종료 메서드가 호출된다.
- build.gradle에 org.springframework.boot:spring-boot-starter-web 라이브러리 추가
→ 스프링 부트는 내장 톰캣 서버를 활용해서 웹서버와 스프링을 함께 실행시킨다.
- 스프링 부트는 웹 라이브러리가 없는 경우, AnnotationConfigApplicationContext를 기반으로 애플리케이션을 구동하고, 있는 경우에는 웹과 관련된 추가 설정과 환경들이 필요하므로 AnnotationConfigServletWebServerApplicationContext를 기반으로 애플리케이션을 구동한다.
- 기본 포트인 8080 포트 중복 시, 포트 변경하기

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter' // spring boot starter  
    //web 라이브러리 추가  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    // JSR-330 Provider  
    implementation 'javax.inject:javax.inject:1'  
    //lombok 라이브러리 추가 시작  
}
```

build.gradle



application.properties

```
1 server.port=9090  
2 |
```

application.properties

session, servletcontext, websocket 범위만 다르지 동작 방식은 비슷하다.

request 스코프

- request 스코프 빈 클래스에 @Scope(value="request") 애노테이션을 붙여준다.
- 스프링 애플리케이션을 실행 시키면 오류가 발생한다.
스프링 애플리케이션을 실행하는 시점에 싱글톤 빈은 생성해서 주입이 가능하지만, request 스코프 빈은 아직 생성되지 않는다. 이 빈은 실제 고객의 요청이 와야 생성할 수 있다.
- 객체 조회를 꼭 필요한 시점까지 지연 처리가 필요하다.
→ Provider or Proxy 방식

request scope 오류 해결 방안 1

Provider

```
@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(); // 고객이 어느 URL로 요청했는지
        MyLogger myLogger = myLoggerProvider.getObject(); // HTTP요청이 진행 중이므로
        System.out.println("myLogger = " + myLogger.getClass()); // request scope 빈 생성
        myLogger.setRequestURL(requestURL);

        myLogger.log("controller test");
        logDemoService.logic(id: "testId");

        return "OK";
    }
}
```

```
@Service
@RequiredArgsConstructor
public class LogDemoService {

    private final ObjectProvider<MyLogger> myLoggerProvider;

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

Proxy

애노테이션으로 원본 객체를 프록시 객체로 대체 가능: 다형성과 DI 컨테이너가 가진 큰 장점

```
@Component
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS) // value 혼자 쓸 때 생략 가능
public class MyLogger {
    private String uuid;
    private String requestURL;
}
```

적용대상이 클래스인 경우 TARGET_CLASS
인터페이스인 경우 INTERFACES
→ MyLogger 가짜 프록시 클래스를 만들어 두고
HTTP request와 상관없이 **가짜 프록시 클래스를 다른 빈에 미리 주입해 둘 수 있다.**

- 가짜 프록시 객체: 요청을 받으면 내부에서 진짜 빈을 요청하는(찾는) 위임 로직을 갖고 있다.
- 스프링 컨테이너는 CGLIB라는 바이트 코드를 조작하는 라이브러리로 MyLogger 클래스를 상속받은 가짜 프록시 객체를 만들어 주입한다.
- 스프링 컨테이너는 myLogger라는 이름으로 진짜 대신 가짜 프록시 객체를 등록한다.
- 의존관계 주입도 가짜 프록시 객체가 주입된다.
- 가짜 프록시 객체는 원본 클래스를 상속 받아서 만들어졌기 때문에 이 객체를 사용하는 클라이언트 입장에서는 사실 원본인지 아닌지 모르게, 동일하게 사용할 수 있다. (다형성)
- request scope와 상관없으며 내부에 단순한 위임 로직으로 싱글톤처럼 동작한다.
- 웹 스코프가 아니어도 프록시는 사용 가능하다.

감사합니다.