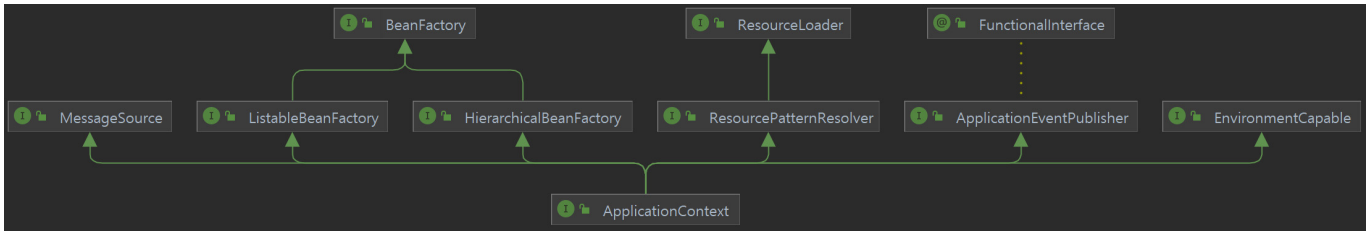
Spring

Java, Spring

BeanFactory, ApplicationContext

1.15. Additional Capabilities of the ApplicationContext



BeanFactory

Docs

- 스프링 최상위 인터페이스로 스프링 빈을 조회/관리하는 역할 담당

ApplicationContext

Docs

- 빈 조회 이외 부가적 기능 제공 (국제화 기능, 리소스 조회 등)

@ComponentScan, @Component

flowchart LR cs["@ComponentScan"] c1["@Component"] c2["@Component"] c3[...] c4["@Component"] cs-. Scan .-> c1 cs-. Scan .-> c2 cs-. Scan .-> c3 cs-. Scan .-> c4

Dependency Injection

- Constructor Injection
- Setter Injection
- Field Injection (Not Recommened. Test가 어려움)
- Method Injection

Lifecycle Callbacks

Docs

`스프링 컨테이너 생성 → 스프링 빈 생성 → DI → 초기화 콜백 → Client가 사용 → 소멸전 콜백 → 스`

```
프링 종료
```

- InitializingBean, DisposableBean Interface를 구현(Impementation).
- @Bean 등록시 “initMethod, destroyMethod” 설정. 기본적으로 destroyMethod는 “close”, “shutdown”이라는 이름의 메서드를 추론하여 자동으로 실행. **외부 라이브러리 사용시 생명 주기를 관리**할 수 있는 유일한 방법.

```
@Bean(initMethod = "init", destroyMethod = "close")
public DBConnection databaseClient() {
  ...
}
```

- 메서드를 구현하고 @PostConstruct, @PostDestroy 사용. (Java 표준) ⇒ 추천!

Bean Scopes

Docs

Bean이 존재할 수 있는 범위

prototype

Client가 Bean 요청시 스프링 DI 컨테이너가 **의존관계**, **초기화**만 관리하여 반환하므로 Life Cycle로 설정한 **종료 메서드를 호출하지 않는다!**

매번 DI 완료된 새로운 Bean이 필요할 때 사용

DI 지연 방법

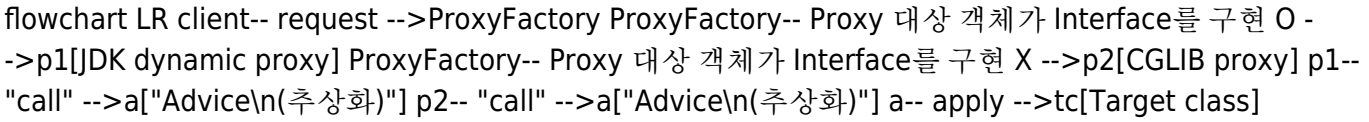
- Provider(ObjectFactory / ObjectProvider)를 DI
- @Scope에서 **proxyMode** 설정. CGLIB를 사용하여 Proxy Pattern을 자동으로 구현해준다!
- 대표적인 예시 : “request” scope. 앱 구동시 아직 request가 들어오지 않은 상태이므로 지연시킬 필요가 있음.

AOP(Aspect Oriented Programming)

- **Cross-cutting Concern을 Module로 분리하는 Programming paradigm.**
  - 개발자가 비지니스 로직에만 집중할 수 있게 한다.
  - 각 프로젝트마다 다른 관심사를 적용할 때 코드의 수정을 최소화 시킬 수 있다.
  - 원하는 관심사의 유지보수가 수월한 코드를 구성할 수 있다.
- 높은 응집도(같은 목적의 Logic 모여있는 정도)를 위해 사용.
- **Spring AOP는 AspectJ 라이브러리의 어노테이션 인터페이스만 사용하고, 동적 런타임중에**

Proxy를 생성하는 방식을 이용한다.

ProxyFactory



- CGLIB는 Proxy 객체를 생성할 때 상속을 이용하기 때문에 Class/Method에 final이 있으면 불가능하다. (Inheritance/Ovrriding 불가)
- ProxyTarget 클래스의 “setProxyTargetClass(true)“를 지정하여 Interface 유무 상관없이 항상 CGLIB로 Proxy 객체를 생성할 수 있다
  - 스프링 부트에서 AOP 적용시 기본값은 true

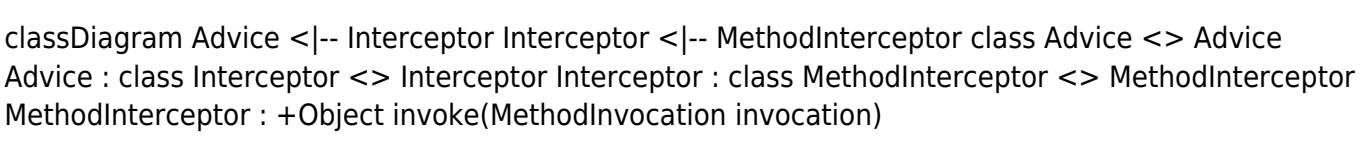
JDK dynamic proxy

- InvocationHandler
- Proxy

CGLIB proxy

외부 라이브러리이나 “org.aopalliance.intercept” 패키지로 편입됨

- Offical docs - How To
- MethodInterceptor



- Spring docs 3.0.x - 8. Spring AOP APIs

Cross-cutting Concern

- 횡단 관심사, 공통관심사항. 대부분의 시스템이 공통으로 가지며 반복되는 보안이나 로그, 트랜잭션 같은 비지니스 로직은 아니지만 **반드시 처리가 필요한 부분**

Joinpoint

- Client가 호출하는 모든 Business Method.
- Pointcut 대상/후보라고도 한다.

Pointcut

- Filtering 된 Joinpoint.
- 수많은 Business Method 중에서 원하는 특정 Method에서만 횡단 관심에 해당하는 공통 기능 수행.
- **Advice가 적용될 위치, Filter Logic**

Advice

- 횡단 관심에 해당하는 공통 기능의 Code. 독립된 Class의 Method로 작성되는 **부가 기능 Logic**
- 해당 기능이 **언제 동작할 지** 스프링 설정 파일에서 지정한다.

종류

- 전체. 첫 번째 인자가 ProceedingJoinPoint이어야 한다. 그리고 이 인자의 “proceed()“가 호출되어야 그 다음 흐름 제어가 가능하다.
  - @Around
- 부분적. 첫 번째 인자가 JoinPoint이어야 한다. “proceed()“가 자동으로 호출된다.
  - @Before : “proceed()” 실행 전
  - @AfterReturning : “proceed()” 실행 후
  - @AfterThrowing : 오류 발생 (try ~ catch의 catch)
  - @After : 모든 작업 수행 후 실행 (try ~ catch의 finally)

실행 순서는 기술한 순서와 같다. 만약 @Around 끼리 순서를 지정할 경우 @Order를 사용한다. 단. @Aspect 기반의 **Class**만 작동한다!

Weaving

- Pointcut으로 지정한 핵심 관심 Method가 호출될 때,
  Advice에 해당하는 횡단 관심 Method가 **삽입되는 과정**
  - 컴파일 시점. ”.class” 파일의 바이트코드를 수정. 별도의 Aspectj 컴파일러 필요
  - 클래스 로딩 시점. ”.class” 파일의 바이트코드를 조작하여 로딩. 'java -javaagent' 옵션 필요
  - 런타임 시점. Proxy 객체 구현 필요
    - 스프링에서 사용하는 방식. 즉, Method 실행 시점에만 Advisor 동작.

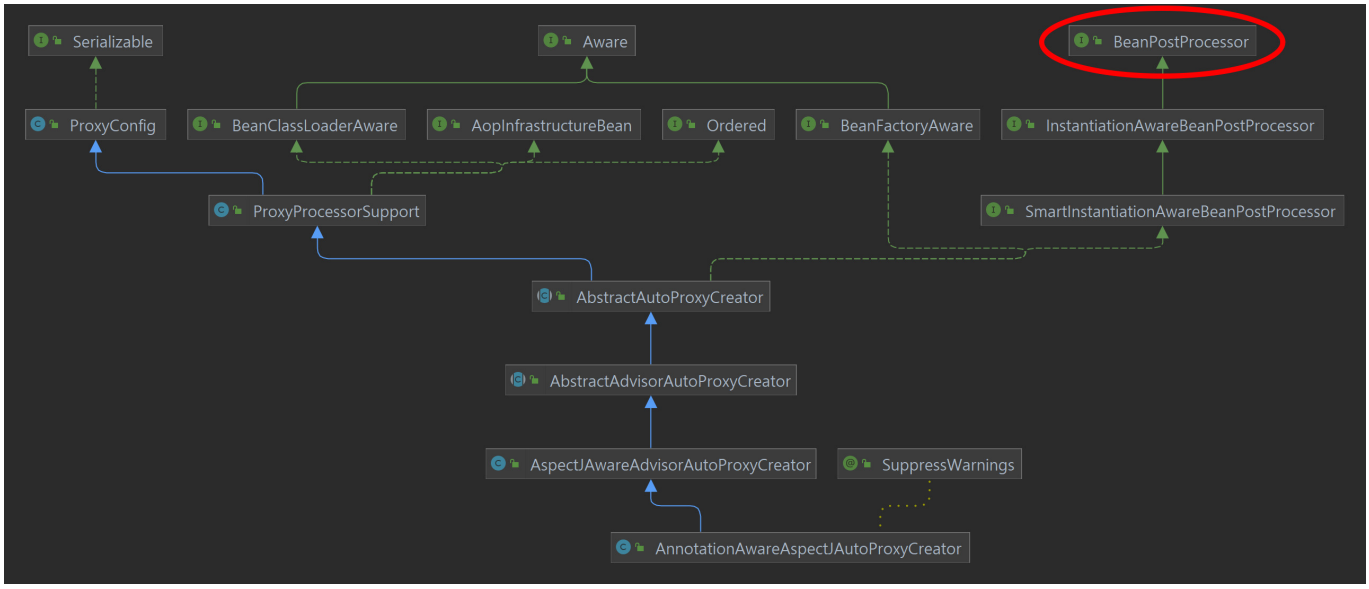
Aspect & Advisor

- 하나의 Pointcut + 하나의 Advice
- Pointcut Method에 **필터링**되어 어떤 Advice Method에 있는 **부가 기능**을 실행할 지 결정.
- 차이점
  - Aspect : Advice 객체의 ID와 Method를 확인할 수 있을 때 사용.
  - Advisor : Advice 객체의 ID나 Method를 확인할 수 없을 때 사용. 대표적인 경우로 Transaction.

Core Concerns

- 핵심 관심. 횡단 관심과 반대 개념. 사용자의 요청에 따라 실제로 수행되는 **핵심 비지니스 로직**

AnnotationAwareAspectjAutoProxyCreator



spring-boot-starter-aop 에 포함된 AnnotationAwareAspectjAutoProxyCreator 빈 후처리기가 작동하여

- @Aspect가 붙은 객체로 등록된 빈을 Advisor로 변환
- 스프링 빈으로 등록된 Advisor 타입의 Bean 탐색

이후 프록시 객체 생성여부를 판단하고
Advisor가 포함된 객체 혹은 일반 객체가 스프링 컨테이너에 등록

용어

POJO(Plain Old Java Object)

- 별도의 API를 사용하지 않은(= 의존성이 없는) 순수 Java Object.

IoC(Inversion of Control, 제어의 역행)

- Method나 Object의 호출 작업을 **개발자가 결정하는 것이 아니라**, 외부(Container)에서 결정되는 것
  - 예를 들어 메인보드에 CPU, RAM, Graphic Card등을 조립하는 데, **각각의 연결이나 호출은 사용자가 아닌 메인보드가 제어.**
  - 즉, **개발자는 부품을 만들어 조립하는 형태의 개발이 가능.**
    - 소스에서 객체 생성과 의존관계에 대한 코드가 사라져 결과적으로 낮은 결합도의 컴포넌트를 구현할 수 있게 된다.

DI(Dependancy Injection, 의존성 주입)

- IoC가 발생할 때 Spring이 내부에 있는 객체(Bean)들의 관계를 관리할 때 사용하는 기법.
- 의존적인 객체를 직접생성하거나, 제어하는 것이 아니라, 제어의 역행으로 특정 객체에 필요한 객체를 외부에서 결정해서 연결시키는 것. (Spring에서 처리)
    - 예를 들어 @Inject 에노테이션 처리된 객체 변수의 객체 생성이나 다른 작업을 개발자가 할 필요가 없음. 해당 부분을 Spring이 맡는다.
- **개발자는 자신이 만드는 객체나 클래스 외에 신경 쓰지 않고 코드를 작성하고, 자신의 코드에 필요한 객체는 Spring을 통해서 주입받는 구조로 작성.**
- 낮은 결합도(의존성)를 위해 사용.

Dependency

- 의존성. POJO와 반대되는 개념. 어떤 객체가 혼자 일을 처리할 수 없는 것을 의미.

Container

- 특정 객체의 생성과 의존 관계 관리를 담당하는 객체

Root Container

- ContextLoaderListener가 생성하는 Spring Container.
    - 가장 먼저 구동되기 때문.

DAO(Data Acess Object)

- Data 접근에 필요한 Method/Object 정의. 보통 DB Table의 **CRUD** 기능 정의. Interface로 제공.
- Spring에서 MyBatis를 호출하고 사용하는 구조로 만들어진다.

Domain Class

- Domain : 개발시 가장 중요한 용어가 될 만한 명사. 여러 물리적인 환경으로 분리가 가능한 하나의 온전한 시스템의 단위.
    - 예) 쇼핑몰 → 회원 / 상품 / 배송
    - 중요한 의미를 가지는 순서에 따라 1차, 2차, 3차로 확대. 이 때, 반드시 필요한 것이 1차 도메인.
- **특정 테이블과 유사한 속성을 가지는 클래스**

VO(Value Object) & DTO(Data Transfer Object)

- 계층간 데이터 교환을 위한 자바빈즈를 말한다. 여기서 말하는 계층간의 컨트롤러, 뷰, 비즈니스 계층, 퍼시스턴스 계층을 말하며 각 계층간 데이터 교환을 위한 객체를 DTO 또는 VO라고 부른다. 즉, Data의 수집과 전달에 사용한다.
- 차이점
    - VO : read only 속성을 가진다. RDB의 Record에 대응하는 Java Class. Database와 가깝다.

    - DTO : 다른 시스템 혹은 레이어간의 데이터 전달 통신 용도로 사용. 메소드 호출 횟수를 줄이기 위해 데이터를 담고 있는 것. 화면에 가깝다.

Command Object

- Controller Method의 Parameter로 받은 VO 객체.
- 요청 Paramter와 Mapping할 **변수**와 **Setter**가 선언되어야 사용 가능.
- Spring Container에서 **자동**으로 값을 Setting해서 넘겨준다.

Criteria Class

- 검색/분류 기준 Class.
- MyBatis의 SQL Mapper를 이용한 DAO 처리시 #{value}의 Property(속성)을 보관하고 Mapper에서 getter()를 제공하는 Class.
- SQL로 넘기는 Parameter가 여러개로 늘어나면 관리가 어렵기 때문에 이를 하나로 묶어서 전달하는 용도로 사용.

Persistence(영속성, 지속성)

- 데이터를 생성한 프로그램의 실행 종료 혹은 컴퓨터가 종료해도 사라지지 않는 데이터의 특성.
- 영속성은 파일 시스템, 관계형 데이터베이스 혹은 객체 데이터베이스 등을 활용하여 구현한다. (일반적으로 DB에 저장하는 것을 의미한다.) 영속성을 갖지 않는 데이터는 단지 메모리에서만 존재하기 때문에 프로그램을 종료하면 모두 잃어버리게 된다. 결국 영속성은 특정 데이터 구조를 이전 상태로 복원할 수 있게 해주어 프로그램의 종료와 재개를 자유롭게 해준다.

Business Layer(비즈니스 계층) & Service

- 고객의 요구사항이 반영되는 영역
- 비즈니스 로직이란 업무에 필요한 데이터 처리를 수행하는 응용프로그램의 일부를 말한다. 이것은 데이터 입력, 수정, 조회 및 보고서 처리 등을 수행하는 루틴, 좀더 엄밀히 말하면 보이는 것의 그 뒤에서 일어나는 각종 처리를 의미한다.
- Controller와 DAO의 접착제 역할. Logic에 필요한 DB 관련 객체들을 모아서 자신이 원하는 일을 처리하는 용도.
- 스프링에서 일반적으로 **Service**라 칭한다.
- 개발 양이 늘어나는데도 사용하는 이유
    - 고객마다 다른 부분을 처리할 수 있는 완충장치 역할. 각 회사마다 다른 Logic이나 규칙을 DB에 무관하게 처리할 수 있는 완충 영역으로 존재할 필요가 있다.
    - Controller와 같은 외부 계층의 호출이 영속 계층에 종속적인 상황을 막아준다. 만약 Controller가 직접 영속 계층의 DB를 이용하게 되면 Transaction의 처리나 Exception의 처리 등 **모든 Logic이 Controller로 집중된다.** 비즈니스 계층은 Controller로 하여금 처리해야 하는 일을 **분업**하게 만들어 준다.

Binding

- Browser에서 들어오는 request가 자동으로 Parameter로 지정한 Class의 객체 속성값으로 처리하는

것

Annotation

의존성 주입시 사용.

Annotation	설명	사용
의존성 관련		
@Qualifier	다수의 Bean 있을 시, 특정 객체의 이름을 이용하여 구분지어 의존성 주입. @Qualifier 끼리 비교 → 이름 → 실패 ※ @Primary보다 우선순위가 높음 (더 상세하게 구현되었기 때문)	Instance variable, Method
@Inject	@Autowired와 동일 기능. (Java에서 제공) 찾는 순서 : 타입 → @Qualifier→ 이름 → 실패	
@Autowired	주로 변수 위에 설정하여 해당 Type의 객체를 찾아서 자동으로 할당. (Spring에서 제공) 찾는 순서 : 타입 → 이름 → @Qualifier → 실패	
@Resource	@Autowired와 @Qualifier 을 합한 기능. (1개, 다수 사용 가능.) 찾는 순서 : 이름 → 타입 → @Qualifier → 실패	
@Value	Properties 값 주입 (사전에 root-context.xml에서 context:property-placeholder 설정 필요)	String variable
Component 관련		
@Component	component scan 설정 후 자동 Bean 생성. 기본적으로 DI 컨테이너가 **Singleton**으로 관리	Class
@Configuration	@Component 기능 및 Bean 설정 정보 관리	
@Primary	같은 타입의 컴포넌트가 있을 시 우선권 명시	
@Controller	@Component 기능 및 Spring MVC의 Controller object 명시 (Web component)	
@Repository	@Component 기능 및 DAO 객체를 Spring에 인식시킴 (Web component)	
@Service	@Component 기능 및 Service 객체 명시. (Web component) 소스를 보면 @Component만 명시되어 있고, 딱히 하는 일이 없음.	
Bean 관련		
@Bean	의존성을 관리할 Bean 생성. @Configuration과 함께 사용시 **Singleton**으로 관리	Method
@Scope	Bean 생명 주기 관리. 기본적으로 Singleton. Docs	
REST 관련		
@ResponseBody	Return type이 HTTP의 응답 메시지로 전송 (객체를 JSON Data로 전송시 사용) (3.0부터 지원)	Method, Return Type
@RestController	View가 아닌 Data 자체를 반환 (jackson-databind와 사용하면 객체 자체를 자동으로 JSON Data로 반환) (4.0부터 지원)	
@PathVariable	현재 URI에서 원하는 Data 추출할 때 사용	Parameter
@RequestBody	request 문자열이 그대로 Paramter로 전달. 전송된 JSON Data를 객체로 변환. @ModelAttribute와 유사하지만 **JSON**에서 사용된다는 점이 차이	

Annotation	설명	사용
@ModelAttribute	자동으로 해당 Command 객체를 View까지 전달. **Parameter에 적용된 경우** : Command 객체의 이름 변경(이 Annotation을 지정하지 않으면 클래스 이름의 첫 글자를 소문자로 변경한 이름이 자동으로 설정.) **Method에 적용된 경우** : 반환된 객체를 View(JSP)에서 사용할 Data 설정. 이 때, @RequestMapping가 적용된 Method보다 먼저 호출된다.	Method, Parameter
@RequestMapping	특정 URI에 Mapping 되는 Class나 Method 명시	Class, Method
@RequestParam	request에서 특정 Parameter 값을 찾아낼 때 사용 (≒ request.getParameter()) Command 객체를 사용하지 못할 경우(Command Class에 없는 Parameter를 추출할 경우) 사용.	Parameter
@RequestHeader	request에서 특정 HTTP 헤더 정보 추출할 때 사용	
@CookieValue	현재 사용자의 쿠키가 존재하는 경우 쿠키의 이름을 이용해서 쿠키 값 추출	
@SessionAttribute	Session상에서 Model의 정보를 유지하고 싶은 경우 사용	Class
@InitBinder	Parameter를 수집해서 객체로 만들 경우에 Customizing	Method

Object

ModelAndView

말그대로 model과 view를 저장하여 반환시켜 사용. 그러나 일반적으로는 Model 사용.

Model

사용할 Data를 addAttribute()로 지정하고, 해당 Data를 보낼 view를 문자열로 반환시켜 사용.

RedirectAttribute

말그래도 “Redirect” 시키며 Attribute 추가.

- addFlashAttribute() : 임시 사용으로 Parameter를 추가하지 않음.
- addAttribute() : Parameter에 추가.

ResponseEntity<T>

상태 코드 제어. AJAX 연동(@RestController)에서 Data를 첨부하여 사용. 특이점으로 상태코드 상관없이 Data는 전송된다.

```
package io.github.ledyx.controller;
```

```
import javax.inject.Inject;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import io.github.ledyx.domain.ReplyVO;
import io.github.ledyx.service.ReplyService;

@RestController
@RequestMapping("/replies")
public class ReplyController {
    @Inject
    private ReplyService service;
    @RequestMapping(value = "/", method = RequestMethod.POST)
    public ResponseEntity<String> add(@RequestBody ReplyVO vo) {
        ResponseEntity<String> entity = null;
        try {
            service.add(vo);
            entity = new ResponseEntity<>("SUCCESS", HttpStatus.OK);
        } catch (Exception e) {
            e.printStackTrace();
            entity = new ResponseEntity<>(e.getMessage(),
HttpStatus.BAD_REQUEST);
        }
        return entity;
    }
}
```