Spring

Contents

I. 일반개념

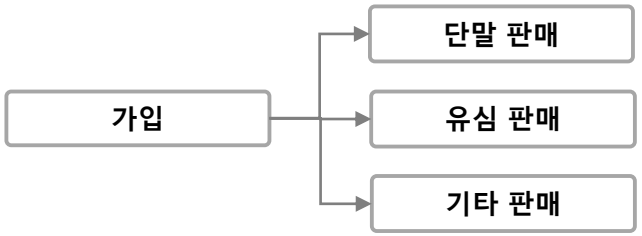
II. Spring

Content

I. 일반개념

1. 결합도

결합도란 하나의 클래스가 다른 클래스와 얼마나 많이 연결되어 있는지를 나타내는 표현으로 결합도가 높으면 유지 보수가 어렵다.



판매는 일반 판매와 할부 판매가 있음

각 판매 Class에 일반 판매, 할부판매 Method 생성

단말 판매	유심 판매	기타 판매
단말일반판매 단말할부판매	유심일반판매 유심할부판매	기타일반판매 기타할부판매

```
@SpringBootApplication
public class Exp01Application {

    public static void main(String[] args) {
        SpringApplication.run(Exp01Application.class, args);

        ModelDevice modelDevice = new ModelDevice();

        modelDevice.modelSale();
        modelDevice.modelInstallment();
    }
}
```

유심 판매로 변경시

- Method Signature가 틀려서 많은 부분이 수정됨
- 여러 Application과 같은 프로그램이 있다면 유지 보수 어려움

```
@SpringBootApplication
public class Exp01Application {

    public static void main(String[] args) {
        SpringApplication.run(Exp01Application.class, args);

        UsimDevice usimDevice = new UsimDevice();

        usimDevice.usimSale();
        usimDevice.usimInstallment();
    }
}
```

```
public class ModelDevice {
    // 일반 판매
    public void modelSale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void modelInstallment() {
        System.out.println("할부  판매");
    }
}
```

```
public class UsimDevice {
    // 일반 판매
    public void usimSale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void usimInstallment() {
        System.out.println("할부  판매");
    }
}
```

```
public class AccessoriesDevice {
    // 일반 판매
    public void accessorieSale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void accessorieInstallment() {
        System.out.println("할부  판매");
    }
}
```

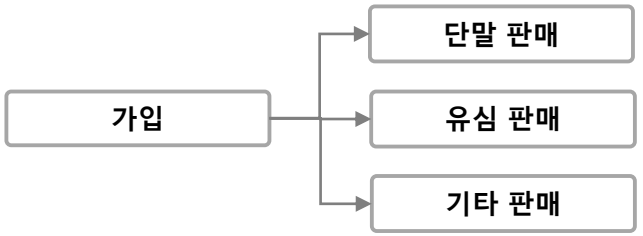
Method를 강제 하려면

- **자바 다양성 이용**
- **디자인 패턴 이용**

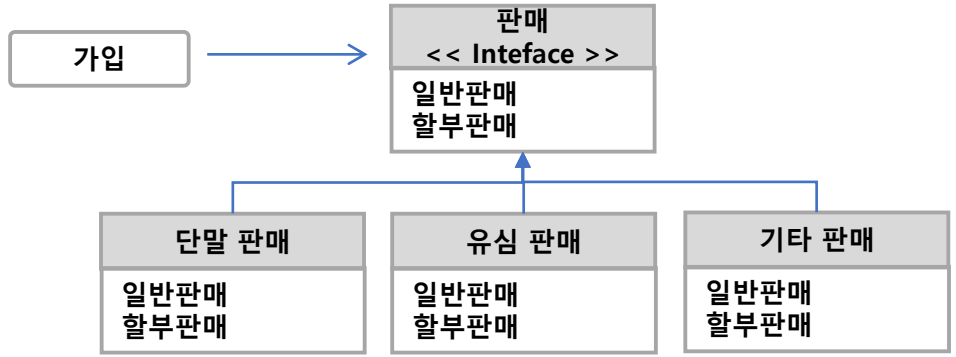
ABACUS

2. 다양성

결합도를 낮추기 위한 방법 중 하나로 객체지향 언어의 다양성을 이용 하여 상속과 메소드 재정의, 형변환을 이용 한다.



판매는 일반 판매와 할부 판매가 있음




```
@SpringBootApplication
public class Exp01Application {
    public static void main(String[] args) {
        SpringApplication.run(Exp01Application.class, args);

        Device device = new ModelDevice();

        device.sale();
        device.installment();
    }
}
```

유심 판매로 변경시

- 업무 로직에는 변경 없이 참조되는 객체만 변경

```
@SpringBootApplication
public class Exp01Application {
    public static void main(String[] args) {
        SpringApplication.run(Exp01Application.class, args);

        Device device = new UsimDevice();

        device.sale();
        device.installment();
    }
}
```

```
public interface Device {
    public void sale();
    public void installment();
}
```

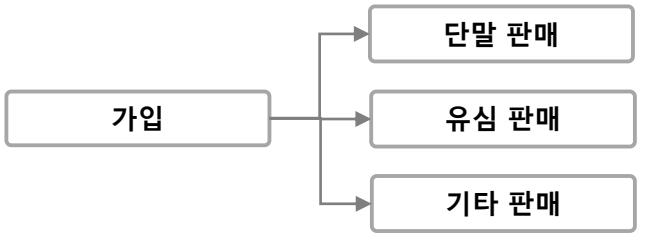
```
public class ModelDevice implements Device {
    // 일반 판매
    public void sale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void installment() {
        System.out.println("할부  판매");
    }
}
```

```
public class UsimDevice implements Device {
    // 일반 판매
    public void sale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void installment() {
        System.out.println("할부  판매");
    }
}
```

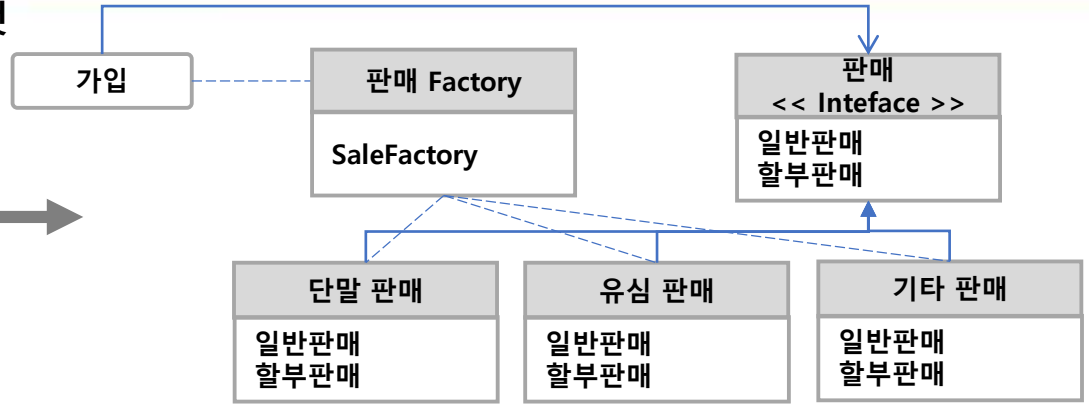
```
public class AccessoriesDevice implements Device {
    // 일반 판매
    public void sale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void installment() {
        System.out.println("할부  판매");
    }
}
```

3. 디자인 패턴 (Factory 패턴)

클라이언트에서 사용할 객체 생성을 캡슐화 하여 느슨한 결합 상태로 만드는 것



판매는 일반 판매와 할부 판매가 있음




```
@SpringBootApplication
public class Exp01Application {
   public static void main(String[] args) {
      SpringApplication.run(Exp01Application.class, args);

         // String saleType = "DEVICE";
         SaleFactory salefactory = new SaleFactory();
         if ( salefactory != null) {
            Device device = (Device) salefactory.getBean(saleType);
            device.sale();
            device.installment();
         }
   }
}
```

유심 판매로 변경시
String saleType = " USIM";

메소드 이면 호출 하는 클라이언트에서
파라메터 변경만 하면 됨

```
public interface Device {
   public void sale();
   public void installment();
}
```

```
public class SaleFactory {
   public Object getBean(String saleType) {
      if ( saleType.equals("DEVICE")) {
         return new ModelDevice();
      } else if (saleType.equals("USIM")) {
         return new UsimDevice();
      } else if (saleType.equals("ASSCESSORIES")) {
         return new AccessoriesDevice();
      }
      return null
   }
}
```

```
public class ModelDevice implements Device {
   // 일반 판매
   public void sale() {
      System.out.println("일반  판매");
   }

   // 할부 판매
   public void installment() {
      System.out.println("할부  판매");
   }
}
```

```
public class UsimDevice implements Device {
   // 일반 판매
   public void sale() {
      System.out.println("일반  판매");
   }

   // 할부 판매
   public void installment() {
      System.out.println("할부  판매");
   }
}
```

```
public class AccessoriesDevice implements Device {
   // 일반 판매
   public void sale() {
      System.out.println("일반  판매");
   }

   // 할부 판매
   public void installment() {
      System.out.println("할부  판매");
   }
}
```

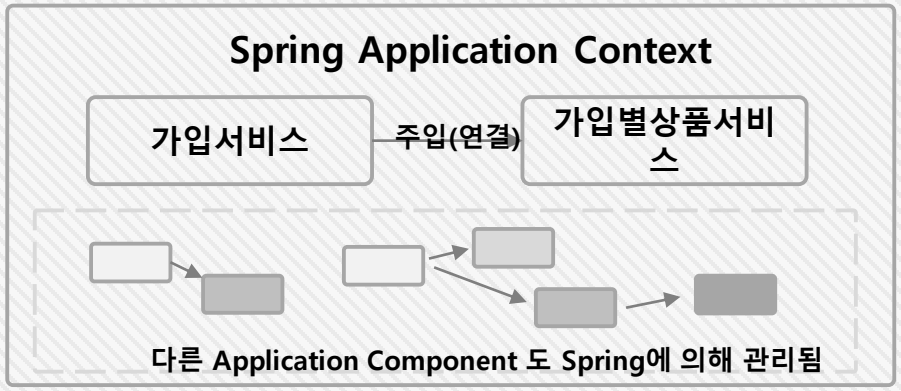
Content

I. Spring

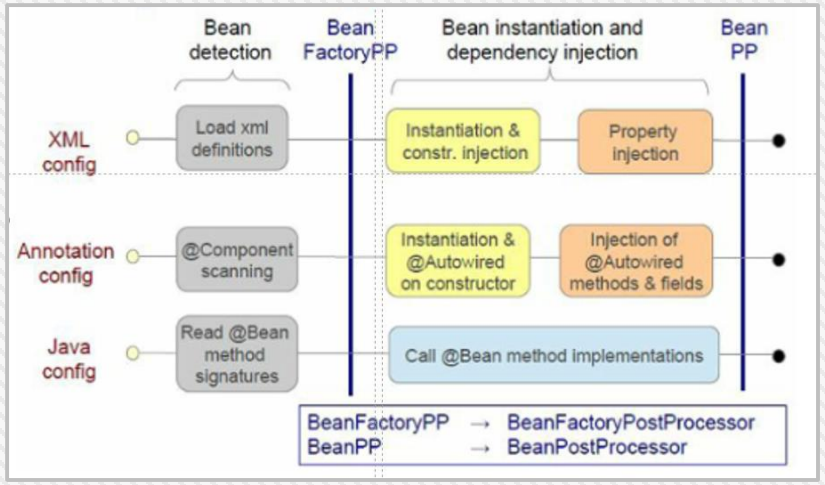
1. 개요

01. Spring ?

- Spring Application Context라는 Container가 Application Component을 생성하고 관리 한다.
- Dependency Injection(DI) 패턴을 기반으로 Bean의 상호 연결을 수행 한다.
  : Application Component에 의존(사용)하는 다른 빈의 생성과 관리를 별도의 Container가 해주며, Component를 필요로 하는 Bean에 주입 한다.

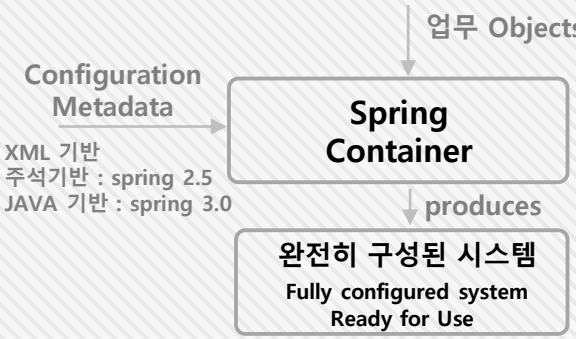


03. Configuration Lifecycle



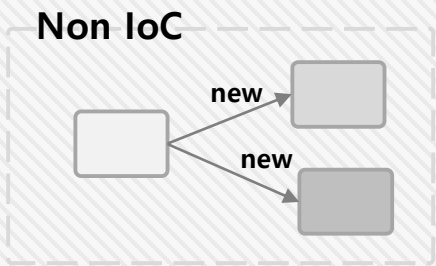
02. Spring IoC(Inversion of Control) Container

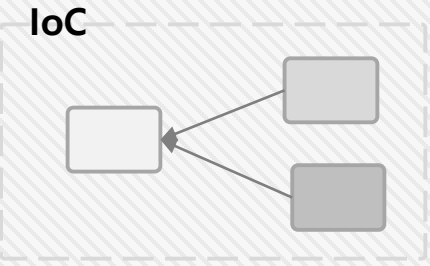
- IoC (inversion of control)
  : 프로그램의 실행 흐름이나 객체의 생명 주기를 개발자가 직접 제어 하는 것이 아니라 컨테이너로 제어권이 넘어가는 것



- Bean
  : 컨테이너가 관리 하는 객체를 의미 하며 기본적으로 싱클턴
  : 기본적으로 네가지 애너테이션을 사용하여 Class를 자동으로 Bean으로 등록

  - ✓ @Controller // Presentation Layer에서 Controller명시
  - ✓ @Service // Business Layer에서 Service 명시
  - ✓ @Repository // Persistence Layer 에서 DAO 명시
  - ✓ @Component // 기타 자동 등록 하고 싶은 것
  - ✓ @Bean // 외부 라이브러리의 객체를 Bean으로 만들때





2. Spring Bean Life Cycle

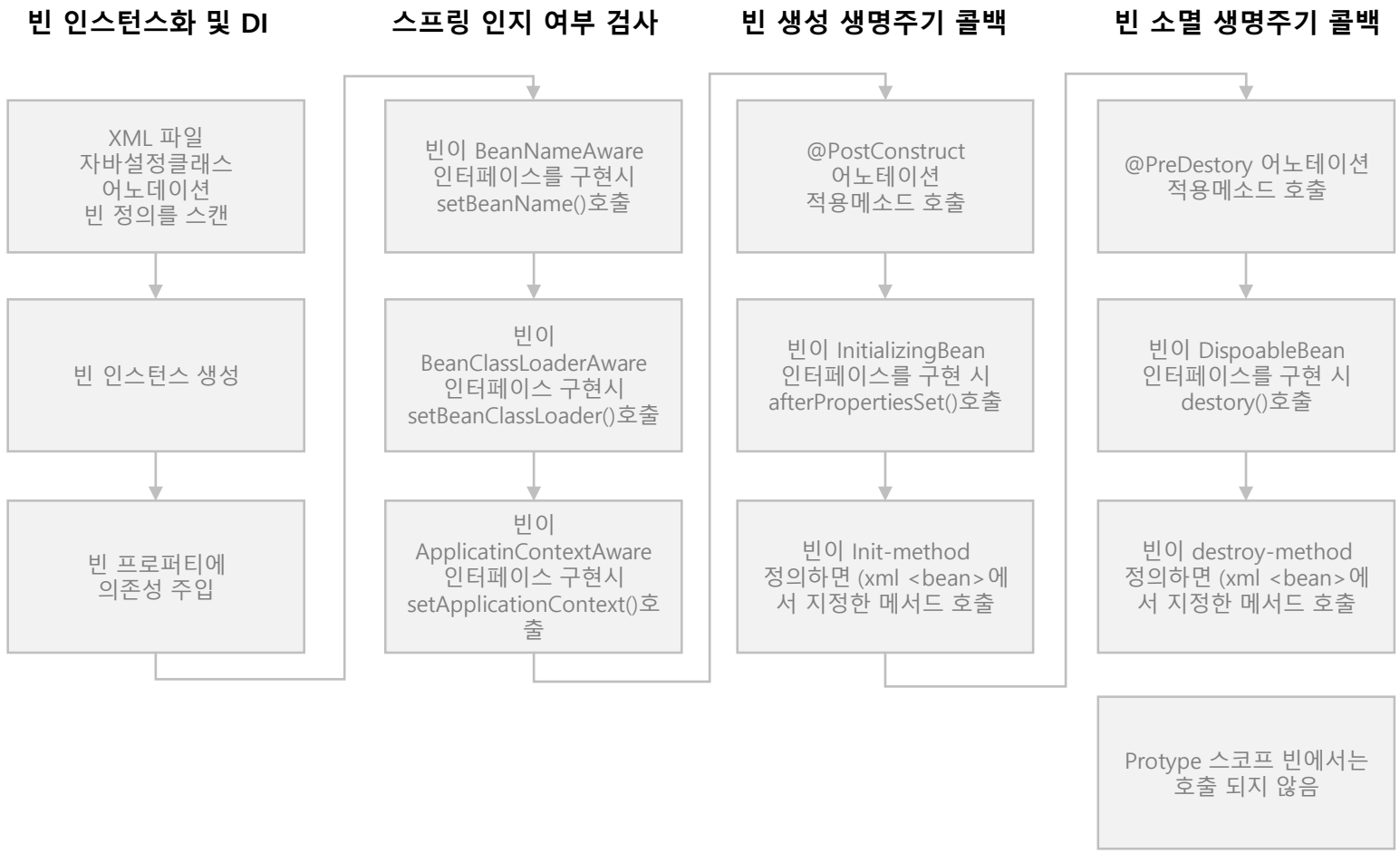
01. Spring Bean Life Cycle

- 인터페이스 구현
  : Spring 에 종속적
- Bean 정의 시 메소드 지정
  : Spring 에 종속적
- JSR-250 어노테이션 지정

```
public class BSimpleClass {

    @PostConstruct
    public void inPostConstructit(){
        System.out.println("BEAN 생성 및 초기화 : init() 호출됨");
    }

    @PreDestroy
    public void destroy(){
        System.out.println("BEAN 생성 소멸 : destroy 호출됨");
    }
}
```

•**@PostConstruct** 어노테이션을 지정한 메소드를 Bean생성과 properties의존성 주입 후 콜백으로 호출

•**@PreDestroy** 어노테이션을 지정한 메소드를 Bean 소멸 전 콜백으로 호출



3. DI (Dependency Injection) - XML

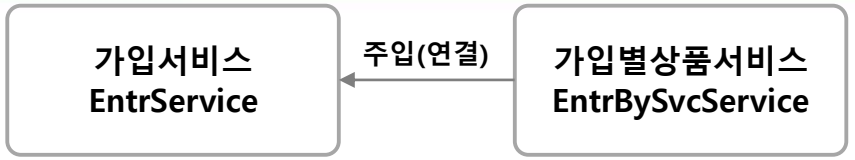
01. DI (Dependency Injection)

- Constructor Injection
  : 생성자에서 받는 방식, final이라 불변(immutable)
- Setter Injection
  : Setter Method를 통해서 주입 해주는 방식

02. XML 기반

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="entrServcice"
        class="co.kr.abacus.spring.xml.entr.service.EntrServiceImpl" >
         <constructor-arg ref="entrBySvcService" />
    </bean>

    <bean id="entrBySvcService"
         class="co.kr.abacus.spring.xml.entrsvc.service.EntrBySvcServiceImpl" />
</beans>
```

```
2020-08-06 01:14:16.127  INFO 58024 --- [           main] o.s.s.concurr
2020-08-06 01:14:16.283  INFO 58024 --- [           main] o.s.b.w.embed
2020-08-06 01:14:16.289  INFO 58024 --- [           main] c.k.abacus.sp
가입 서비스
상품 서비스
```



#. 가입별상품서비스 bean을 가입서비스 bean에 연결

```
@SpringBootApplication
@ImportResource({"classpath*:applicationContext.xml"})
public class ExpXmlApplication {

    public static void main(String[] args) {
        ApplicationContext applicationContext = SpringApplication.run(ExpXmlApplication.class, args);
        EntrService entrService = applicationContext.getBean(EntrService.class);
        entrService.entrSvc();
    }
}
```

```
public interface EntrService {
    public void entrSvc();
}
```

```
public class EntrServiceImpl implements EntrService {
    private EntrBySvcService entrBySvcServcie;
    public EntrServiceImpl(EntrBySvcService entrBySvcServcie) {
        this.entrBySvcServcie = entrBySvcServcie;
    }
    @Override
    public void entrSvc() {
        System.out.println("가입 서비스");
        entrBySvcServcie.entrBySvc();
    }
}
```

```
public interface
EntrBySvcService {
    public void entrBySvc();
}
```

```
public class EntrBySvcServiceImpl implements EntrBySvcService {
    @Override
    public void entrBySvc() {
        System.out.println("상품 서비스");
    }
}
```

4. DI (Dependency Injection) - annotation

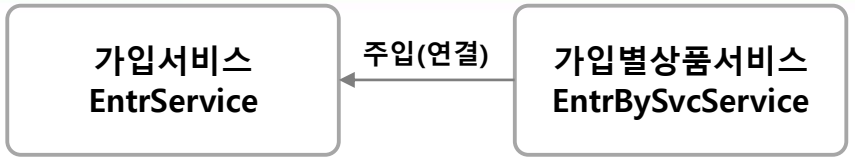
01. 컴포넌트 스캔 설정 (component-scan)

- 스프링 설정 파일에 Application에서 사용할 Bean을 등록 하지 않고 자동 설정
- <context:component-scan> element 정의
- Class에 @Component 설정

02. XML 기반

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

   <context:component-scan
        base-package="co.kr.abacus.spring.annotation"></context:component-scan>


</beans>
```

```
2020-08-06 01:55:35.133  INFO 49560 --- [           main] c.k.a.s
2020-08-06 01:55:35.137  INFO 49560 --- [           main] c.k.a.s
2020-08-06 01:55:35.575  INFO 49560 --- [           main] c.k.a.s
가입 서비스
상품 서비스
```



#. 가입별상품서비스 bean을 가입서비스 bean에 연결

```
@SpringBootApplication
@ImportResource({"classpath*:applicationContext.xml"})
public class ExpAnnotationApplication {
   public static void main(String[] args) {
      ApplicationContext applicationContext = SpringApplication.run(ExpAnnotationApplication.class, args);

      EntrService entrService = (EntrService) applicationContext.getBean("EntrService");
      entrService.entrServcie();
   }
}
```

```
public interface EntrService {
   public void entrSvc();
}
```

```
@Component("EntrService")
public class EntrServiceImpl implements EntrService {
   private EntrBySvcService entrBySvcServcie;
   public EntrServiceImpl(EntrBySvcService entrBySvcServcie) {
      this.entrBySvcServcie = entrBySvcServcie;
   }
   @Override
   public void entrSvc() {
      System.out.println("가입 서비스");
      entrBySvcServcie.entrBySvc();
   }
}
```

```
public interface EntrBySvcService
{
   public void entrBySvc();
}
```

```
@Component("entrBySvcService")
public class EntrBySvcServiceImpl implements EntrBySvcService {
   @Override
   public void entrBySvc() {
      System.out.println("상품 서비스");
   }
}
```

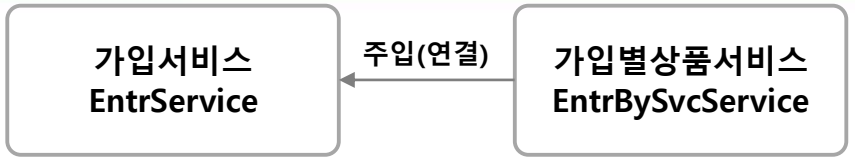
5. DI (Dependency Injection) – JAVA Config

01. 자바 설정 기반

- 스프링 설정 파일에 Application에서 사용할 Bean을 등록 하지 않고 자동 설정
- @Configuration 사용

02. 자바 Config 파일

```
@Configuration
public class ServiceConfig {

   @Bean
   public EntrService entrService() {
      return new EntrServiceImpl(entrBySvcService());
   }

   @Bean
   public EntrBySvcService entrBySvcService() {
      return new EntrBySvcServiceImpl();
   }              }
}
```



#. 가입별상품서비스 bean을 가입서비스 bean에 연결

```
@SpringBootApplication
@ImportResource({"classpath*:applicationContext.xml"})
public class ExpAnnotationApplication {
   public static void main(String[] args) {
      ApplicationContext applicationContext = SpringApplication.run(ExpAnnotationApplication.class, args);

      EntrService entrService = (EntrService) applicationContext.getBean("entrService");
      entrService.entrServcie();
   }
}
```

```
public interface EntrService {
   public void entrSvc();
}
```

```
public class EntrServiceImpl implements EntrService {
   private EntrBySvcService entrBySvcServcie;

   public EntrServiceImpl(EntrBySvcService entrBySvcServcie) {
      this.entrBySvcServcie = entrBySvcServcie;
   }

   @Override
   public void entrSvc() {
      System.out.println("가입 서비스");
      entrBySvcServcie.entrBySvc();
   }
}
```

```
public interface EntrBySvcService
{
   public void entrBySvc();
}
```

```
public class EntrBySvcServiceImpl implements EntrBySvcService {
   @Override
   public void entrBySvc() {
     System.out.println("상품 서비스");
   }
}
```

6. DI (Dependency Injection) – 의존성 주입

Spring에서 의존성 주입을 지원하는 어노테이션은 @Autowired, @Inject, @Qualifier, @Resource 있음, Spring에서는 @Autowired, @Qualifier 제공 함

01. @Component
- Component-scan 대상이 되는 객체 -> Spring에서 기능에 따라 추가 제공

02. @Autowired
- 생성자, 메소드, 멤버변수 위에 모두 사용 가능 하나 주로 멤버변수 위에 선언
  Spring Container는 멤버변수의 타입을 체크 하여 해당 객체를 변수에 주입
- @Inject은 동일한 기능 임

03. @Qualifier
- 의존성 대상이 되는 한 개 이상인 경우 지정

04. @Resource
- 객체의 이름을 이용하여 의존성 주입

05. Spring 제공
- @Service
  : 비즈니스 로직
- @Repository
  : DB 연동
- @Controller
  : 사용자 요청

```
@SpringBootApplication
public class ExpDiApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            ExpDiApplication.class, args);

    }
}
```

```
@Component
public class ExpDiRunner
        implements ApplicationRunner {

    @Autowired
    private EntrService entrServcie;

    @Override
    public void run(ApplicationArguments args)
            throws Exception {
        entrServcie.entrServcie();

    }
}
```

```
public interface EntrService {
    public void entrSvc();
}
```

```
@Component
public class EntrServiceImpl implements EntrService {

    @Autowired
    private EntrBySvcService entrBySvcServcie;

    @Autowired
    @Qualifier("modelDevice")
    private Device modelDevice;


    @Resource(name = "usimDevice")
    private Device usimDevice;


    @Override
    public void entrServcie() {
        entrBySvcServcie.entrBySvc();
        modelDevice.sale();
        usimDevice.sale();
    }
}
```

```
public interface
EntrBySvcService {
    public void entrBySvc();
}
```

```
@Component
public class EntrBySvcServiceImpl implements EntrBySvcService {
    @Override
    public void entrBySvc() {
        System.out.println("상품 서비스");
    }
}
```

```
public interface Device {
    public void sale();
    public void installment();
}
```

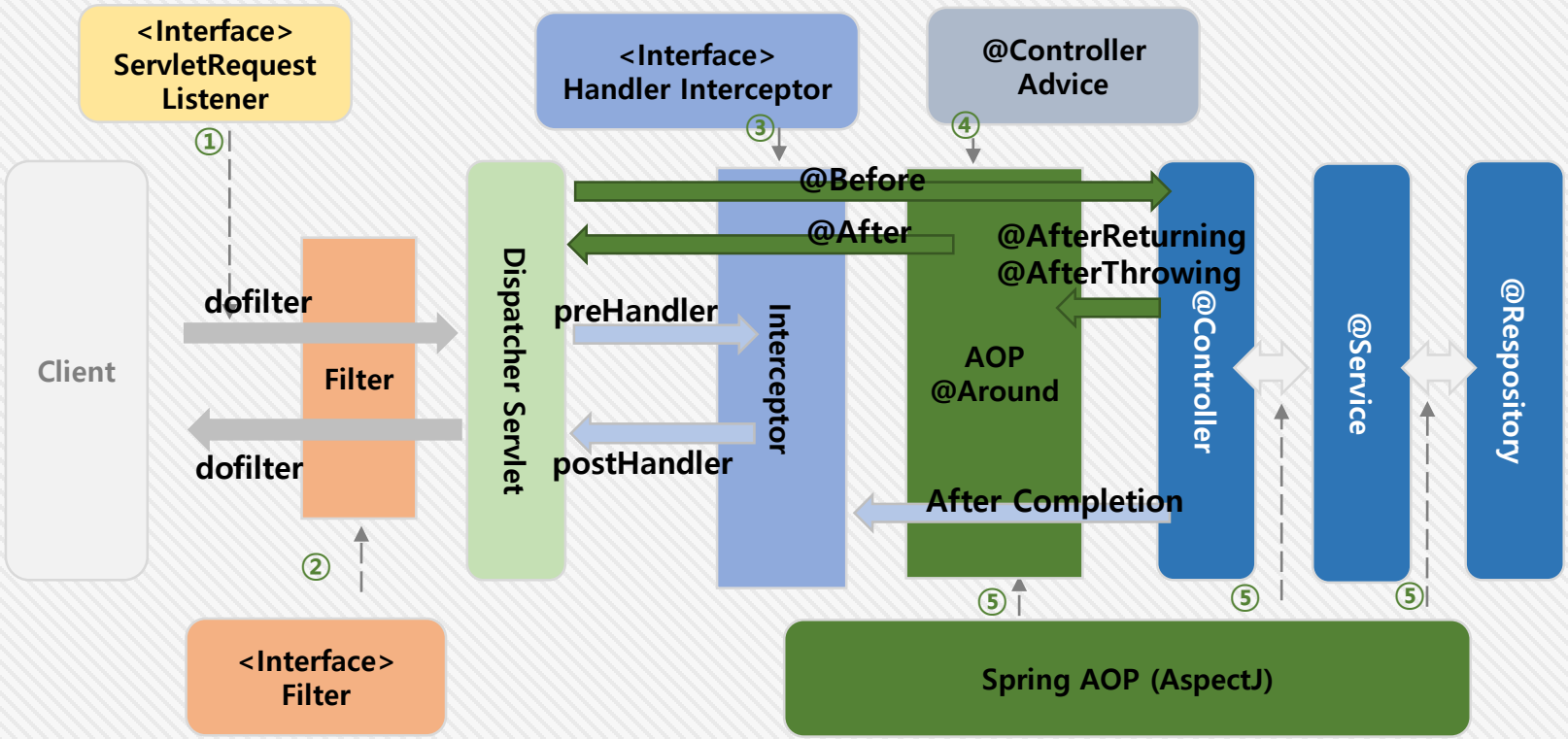
```
@Component("modelDevice")
public class ModelDevice implements Device {
    // 일반 판매
    public void sale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void installment() {
        System.out.println("할부  판매");
    }
}
```

```
@Component("usimDevice")
public class UsimDevice implements Device {
    // 일반 판매
    public void sale() {
        System.out.println("일반  판매");
    }

    // 할부 판매
    public void installment() {
        System.out.println("할부  판매");
    }
}
```

7. Filter, Interceptor, AOP

01. Filter, Interceptor, AOP 흐름

- Interceptor와 Filter는 Servlet 단위에서 실행된다. <> 반면 AOP는 메소드 앞에 Proxy패턴의 형태로 실행 됨
- 요청이 들어오면 Filter → Interceptor → AOP → Interceptor → Filter 순으로 거치게 된다
- Filter
  : 요청과 응답 사이에서 Data 정체 역할
  : 스프링 컨텍스트 외부에 존재하여 스프링과 무관한 자원에 대해 동작 - 예) 인코딩, XSS 방어 등
  : Fliter 실행 메소드
  - init() - 필터 인스턴스 초기화
  - doFilter() - 전/후 처리
  - destroy() - 필터 인스턴스 종료
- Interceptor
  : 인터셉터는 스프링의 DistpatcherServlet이 컨트롤러를 호출하기 전, 후로 끼어들기 때문에 스프링 컨텍스트(Context, 영역) 내부에서 Controller(Handler)에 관한 요청과 응답에 대해 처리
  : 스프링의 모든 빈 객체에 접근
  : HttpServletRequest, HttpServletResponse를 파라미터로 사용
  : 인터셉터는 여러 개를 사용할 수 있고 로그인 체크, 권한체크, 프로그램 실행시간 계산작업 로그확인 등의 업무처리
  : 인터셉터의 실행 메서드
  - preHandler() - 컨트롤러 메서드가 실행되기 전
  - postHanler() - 컨트롤러 메서드 실행직 후 view페이지 렌더링 되기 전
  - afterCompletion() - view페이지가 렌더링 되고 난 후
- AOP
  : 메소드 전후의 지점에 자유롭게 설정
  : Advice의 경우 JoinPoint나 ProceedingJoinPoint 등을 활용
  : AOP의 포인트컷
  - @Before: 대상 메서드의 수행 전
  - @After: 대상 메서드의 수행 후
  - @After-returning: 대상 메서드의 정상적인 수행 후
  - @After-throwing: 예외발생 후
  - @Around: 대상 메서드의 수행 전/후



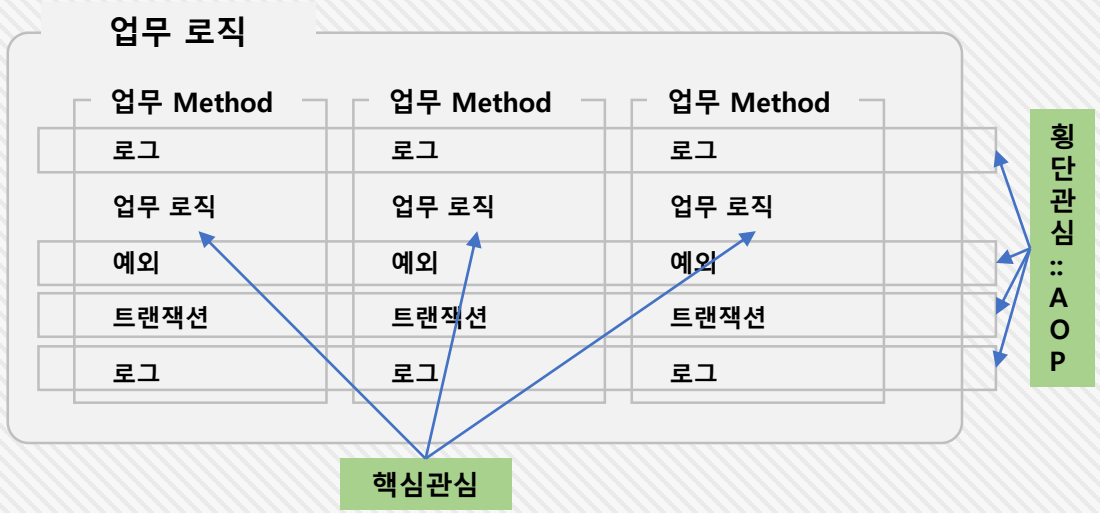
- javax.servlet.Filter.ServletRequestListener
  : 요청 시작과 요청 종료시의 타이밍에서 어떤 작업을 수행
- javax.servlet.Filter
  : **Servlet, JSP, 정적 콘텐츠 등의 Web 리소스에 대한 액세스 전후에 공통 작업을 수행**
- HandlerInterceptor
  : **Spring MVC의 Handler의 호출 전후에 일반적인 작업을 수행**
- @ControllerAdvice
  : Controller 전용의 특수한 메소드 (@InitBainder메소드,@ModelAttribute메소드,@ExceptionHandler메소드)를 복수의 Controller에서 공유
- Spring AOP (AspectJ)
  : **Spring의 DI 컨테이너에서 관리되는 Bean의 public 메소드 호출 전후에 일반적인 작업을 수행**

8. AOP

DI(Dependency Inject)가 결합도를 낮추는 것이라면 AOP은 응집도와 관련된 기능으로 횡단 관심 분리임.

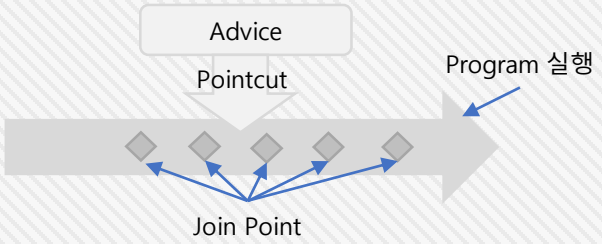
01. 횡단 관심과 핵심 관심 분리

- 메소드 마다 공통인 로깅, 예외, 트랜잭션과 같은 횡단 관심과 실제 수행 되는 업무 로직을 핵심 관심이라 함



02. AOP 용어

- Advice : **Aspect가 해야 할 작업으로 언제 무엇을 해야 하는지 정의**
  - before(이전) : advice 대상 Method가 호출 되기 전
  - after(이후) : 결과에 상관없이 advice 대상 Method사 완료 된 후
  - after-returning(반환 이후) : advice 대상 Method가 성공적으로 완료 된 후
  - after-throwing(예외 발생 이후) : advice 대상 Method가 예외를 던진 후
  - around(주위) : advice가 advice 대상 메소드를 감싸서 advice 대상 Method 호출 전과 호출 후 몇가지 기능 제공
- Join point : **Advice를 적용 할 수 있는 곳으로** Application 실행에 Aspect를 끼워 넣을 수 있는 지점, **모든 업무 메소드**
- Pointcut : Aspect가 Advice할 Join point의 영역을 좁히는 일을 함, 즉 **어디서 할지를 정의 함**
  - 각 pojntcut은 Advice가 weaving되어야 하나 이상의 Join point를 정의함
- Aspect : Advice와 Pointcut를 합친 것으로 무엇을 언제 어디서 할지 모든 정보를 정의함
- Introduction : 기존 Class에 코드 변경 없이도 새 Method, Member Variable을 추가 하는 기능
- Weaving : 타깃 객체에 Aspect를 적용해서 새로운 프록시 객체를 생성하는 절차
  - compile time : 컴파일 시점에 weaving. 별도의 컴파일러 필요, Aspect 5의 Weaving Compile
  - classload time : JVM에 로드 될 때 weaving. AspectJ 5의 Load-Time Weaving 기능 사용
  - runtime : 실행 중에 weaving. Spring AOP Aspect

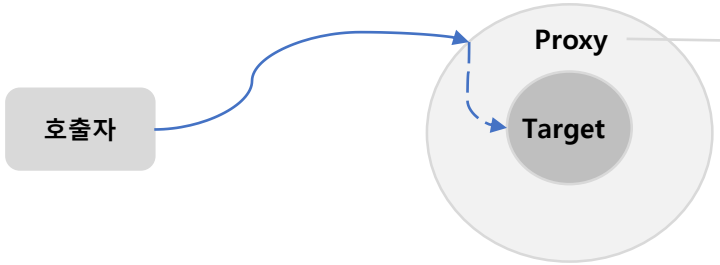


03. Spring의 AOP

- Classic Spring Proxy 기반 AOP
- Pure-POJO Aspect
- @AspectJ Annotation 기반 Aspect
- AspectJ Aspect에 bean 주입

9. Sprig AOP

Spring Aspect는 Target 객체를 감싸는 프록시 형태로 구현되며, 이 프록시는 먼저 호출을 가로챈 후 추가적인 Aspect 로직을 수행 하고 나서야 Target Method를 호출 한다.



실행 시간에 생성 -> Aspect가 Spring 관리 Bean weaving

동적 프록시를 기반으로 AOP를 구현 하므로 메소드 Join Point만 지원함

01. Pointcut 표현식

- execution : 메소드 실행 Join Point와 일치 시키는데 사용
- within : 특정 타입에 속하는 Join Point 정의
- bean : bean 이름으로 pointcut

```
package co.kr.abacus

public interface SvcByEntrService {
    public void saveService
}
```

```
                ③  ④                    ⑤       ⑥
Execution ( * co.kr.abacus.SvcByEntrService.saveService(...) )
①          ②
```

① 메소드 실행 시작 ② 메소드 명세

③ 리턴 타입 지정

- * : 모든 리턴 타입 허용
- void : 리턴 타입이 void인 메소드
- !void : 리턴 타입이 void가 아닌 메소드

④ 메소드가 속하는 타입

- 패키지 + 클래스
- 패키지 지정
  - co.kr.abacus : 해당 패키지만
  - co.kr.abacus.. : 지정된 패키지로 시작하는 모든 패키지
- 클래스
  - FullName (SvcByEntrService) : 헤당 클래스만
  - *Service : 이름이 Service로 끝나는 클래스
  - Service+: 클래스 이름 뒤에 '+'가 붙으면 해당 클래스로부터 파생된 모든 자식 클래스 선택, 인터페이스 이름 뒤에 '+'가 붙으면 해당 인터페이스를 구현한 모든 클래스

⑤ 메소드

- * : 모든 메소드 선택
- save* : save로 시작 하는 모든 메소드

⑥ 인자

- (..) : 모든 매개변수
- (*) : 반드시 1개의 매개변수를 가지는 메소드만 선택
- (Fullpackage) : 매개변수로 작성된 Class가 가지고 있는 메소드
- (!Fullpackage) : 매개변수로 작성된 Class를 가지지 않는 메소드
- (Integer, ..) : 한 개 이상의 매개변수를 가지되, 첫 번째 매개변수의 타입이 Integer인 메소드만
- (Integer, *) : 반드시 두 개의 매개변수를 가지되, 첫 번째 매개변수의 타입이 Integer인 메소드만 -

#. 범위 제한

```
Execution ( * co.kr.abacus.SvcByEntrService.saveService(...))
      && within(co.kr.abacus.* )
      ⑦      ⑧
```

⑦ 조합 및 연산자

- && : and 연산자
- || : or 연산자
- ! : 부정

ABACUS

10. Sprig AOP – JoinPoint Interface

Advice 메소드를 의미 있게 구현하려면 클라이언트가 호출한 비즈니스 메소드의 정보가 필요하다. 예를 들면 예외가 터졌는데, 예외발생한 메소드의 이름 등을 기록할 필요가 있을 수 있다. 이럴때 JoinPoint 인터페이스가 제공하는 유용한 API들이 있다

Signature getSignature() : 클라이언트가 호출한 메소드의 시그니처(리턴타입, 이름, 매개변수) 정보가 저장된 Signature 객체 리턴

Object getTarget() : 클라이언트가 호출한 비즈니스 메소드를 포함하는 비즈니스 객체 리턴

Object[] getArgs() : 클라이언트가 메소드를 호출할 때 넘겨준 인자 목록을 Object 배열 로 리턴

String getName() : 클라이언트가 호출한 메소드 이름 리턴

String toLongString() : 클라이언트가 호출한 메소드의 리턴타입, 이름, 매개변수(시그니처)를 패키지 경로까지 포함 하여 리턴

String toShortString() : 클라이언트가 호출한 메소드 시그니처를 축약한 문자열로 리턴

```
    @Around("entrBySvc()")
    public void watchEntrBySvc(ProceedingJoinPoint  pjp) {
        logger.info("start - " + pjp.getSignature().getDeclaringTypeName() + " / " + pjp.getSignature().getName());
        try {
            pjp.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        logger.info("finished - " + pjp.getSignature().getDeclaringTypeName() + " / " + pjp.getSignature().getName());
    }
}
```

11. Sprig AOP – 예제

#. pom.xml

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

#. Application Start

```
@EnableAspectJAutoProxy
@SpringBootApplication
public class ExpAopApplication {
   public static void main(String[] args) {
      SpringApplication.run(ExpAopApplication.class, args);
   }
}
```

DI 의존성 주입

```
@Component
public class ExpAopRunner implements ApplicationRunner {

   @Autowired
   private EntrBySvcService entrBySvcService;

    @Autowired
    private EntrService entrService;


    @Override
    public void run(ApplicationArguments args)
          throws Exception {
       System.out.println("*** Spring Autowired *************");
       entrService.entrServcie();
       entrBySvcService.entrBySvc();
    }
}
```

```
@Aspect
@Component
public class EntrBySvcAspect {
   Logger logger = LoggerFactory.getLogger(EntrBySvcAspect.class);
   // Pointcut 정의
   @Pointcut("execution(* co.kr.abacus.spring.aop.entrsvc.service.EntrBySvcService.entrBySvc(..))")
   public void entrBySvc() {};

   //@Before("execution(* co.kr.abacus.spring.aop.entrsvc.service.EntrBySvcService.entrBySvc(..))")
   @Before("entrBySvc()")
    public void beforeService() { logger.info("*** 실행 이전 "); }
   @AfterReturning("entrBySvc()")
   public void afterReturningService() { logger.info("*** 실행 성공  "); }
   @AfterThrowing("entrBySvc()")
   public void AfterThrowingService() { logger.info("*** 실행 실패  "); }
   @Around("entrBySvc()")
   public void watchEntrBySvc(ProceedingJoinPoint  pjp) {
      logger.info("start - " + pjp.getSignature().getDeclaringTypeName() + " / " + pjp.getSignature().getName());
      try {
         pjp.proceed();
      } catch (Throwable e) {
         e.printStackTrace();
      }
      logger.info("finished - " + pjp.getSignature().getDeclaringTypeName() + " / " + pjp.getSignature().getName());
   }
}
```

```
@Configuration
public class EntrBySvcConfig {
    @Bean
    public EntrServiceAspect entrBySvcBeanAspect() {
       return new EntrServiceAspect();
    }
}
```

Bean 주입

```
@Aspect
public class EntrServiceAspect {
   Logger logger = LoggerFactory.getLogger(EntrServiceAspect.class);
....
}
```

```
*** Spring Autowired *************
2020-08-08 01:58:03.497  INFO 63752 --- [           main] c.k.a.s.aop.aspect.EntrServiceAspect     : Bean start - co.kr.abacus.spring.aop.entr.service.EntrServiceImpl / entrServcie
2020-08-08 01:58:03.497  INFO 63752 --- [           main] c.k.a.s.aop.aspect.EntrServiceAspect     : *** Bean 실행 이전
가입 서비스
2020-08-08 01:58:03.501  INFO 63752 --- [           main] c.k.a.s.aop.aspect.EntrServiceAspect     : *** Bean 실행 성공
2020-08-08 01:58:03.501  INFO 63752 --- [           main] c.k.a.s.aop.aspect.EntrServiceAspect     : Bean finished - co.kr.abacus.spring.aop.entr.service.EntrServiceImpl / entrServcie
2020-08-08 01:58:03.501  INFO 63752 --- [           main] c.k.a.spring.aop.aspect.EntrBySvcAspect  : start - co.kr.abacus.spring.aop.entrsvc.service.EntrBySvcServiceImpl / entrBySvc
2020-08-08 01:58:03.501  INFO 63752 --- [           main] c.k.a.spring.aop.aspect.EntrBySvcAspect  : *** 실행 이전
상품 서비스
2020-08-08 01:58:03.503  INFO 63752 --- [           main] c.k.a.spring.aop.aspect.EntrBySvcAspect  : *** 실행 성공
2020-08-08 01:58:03.503  INFO 63752 --- [           main] c.k.a.spring.aop.aspect.EntrBySvcAspect  : finished - co.kr.abacus.spring.aop.entrsvc.service.EntrBySvcServiceImpl / entrBySvc
```

12. Filter

01. servlet.Filter

- javax.servlet-api나 tomcat-embed-core를 사용하면 제공되는 servlet filter interface

```
@Order(1)
@Component
public class ServletFilter implements Filter {
  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
          throws IOException, ServletException {
     chain.doFilter(request, response);
   }
}
```

02. Spring에서 제공 하는 GenericFilterBean

```
@Order(3)
public class GenericFilterBeanFlter extends GenericFilterBean {
   @Override
   public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
          throws IOException, ServletException {
      // TODO 전처리
      chain.doFilter(request, response);
   }
}

@Configuration
public class GenericFilterBeanFilterConfig {
   @Bean
    public GenericFilterBeanFlter GenericFilterBeanFlte() {
       return new GenericFilterBeanFlter();
    }
}
```

03. Spring에서 제공 하는 OncePerRequestFilter

```
@Order(0)
@Component
public class SomeFilter extends OncePerRequestFilter{
   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
          throws ServletException, IOException {
     chain.doFilter(request, response);
   }
}
```

13. Interceptor

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

@Autowired
private SomeInterceptor someInterceptor;

@Override
public void addInterceptors(InterceptorRegistry registry) {
registry.addInterceptor(someInterceptor)
      .addPathPatterns("/some/**")
      .excludePathPatterns("/etc/**");
}
}
```

```
@Component
public class SomeInterceptor extends HandlerInterceptorAdapter {

Logger logger = LoggerFactory.getLogger(SomeInterceptor.class);

@Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler){
logger.info("===== before(interceptor) =====");
        return true;
    }

    @Override
    public void postHandle(
            HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)
            throws Exception {
    logger.info("===== after(interceptor) =====");
    }

    @Override
    public void afterCompletion(
            HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
            throws Exception {
    logger.info("===== afterCompletion =====");
    }
}
```

http://localhost:8080/some

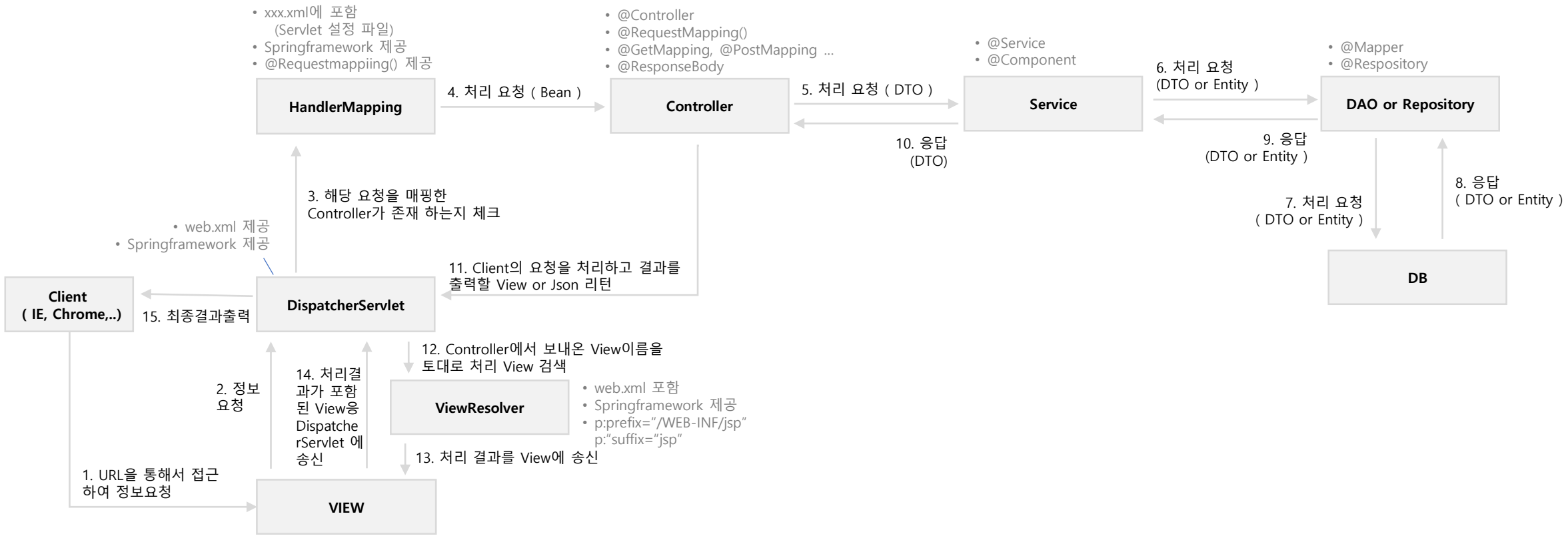
```
c.k.a.s.f.interceptor.SomeInterceptor    : ===== before(interceptor) =====
c.k.a.s.f.controller.SomeController      : ***** GetMapping ******
c.k.a.s.f.interceptor.SomeInterceptor    : ===== after(interceptor) =====
c.k.a.s.f.interceptor.SomeInterceptor    : ===== afterCompletion =====
```

http://localhost:8080/etc

```
c.k.a.s.f.controller.SomeController      : ***** GetMapping ******
```

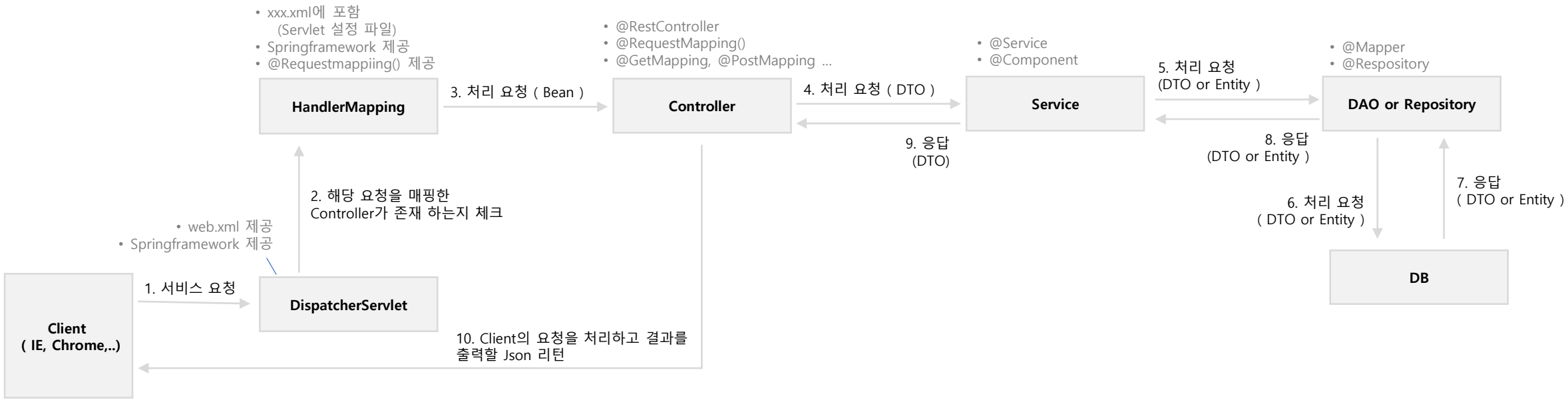
14. Spring MVC - @Controller

Spring MVC 구조



15. Spring MVC - @RestController

Spring MVC 구조 : @RestController = @Controller + @ResponseBody



16. Spring MVC – 사용하는 Annotation

01. @RequestBody, @ResponseBody

- @RequestBody
  : Http요청의 Body내용을 자바 객체로 매핑
- @ResponseBody
  : 자바 객체를 Http 응답의 Body내용으로 매핑 (Json)

```
@RestController
Public class ServiceController {
   // Http 요청의 내용을 RequestDTO 객체에 매핑하기 위해서 @RequestBody 사용
   // @ResponseBody를 사용 하지 않는 이유 : @RestController 사용 하였기 때문
   // @Controller를 사용 하는 경우 :  @ResponseBody를 사용 해야 함
   @RequestMapping(value="/uri/process", method = RequestMethod.POST)
   public ResponseDTO process(@RequestBody RequestDTO requestDTO) {
      ResponseDTO responseDTO = service.process(requestDTO);
      return responseDTO;
   }
}
```

02. @RequestMappiing Method

- GET
  : 요청 받은 URI의 정보를 검색
- POST
  : 요청된 자원을 생성
- PUT
  : 요청된 자원을 수정 , 요청된 자원 전체 갱신
- PATCH
  : 요청된 자원을 수정, 일부만 갱신
- DELETE
  : 요청된 자원 삭제
- OPTIONS
  : 지원 되는 메소드의 종류 확인

02. @RequestMappiing

- Uri을 Controller에 매핑 해 주는 기능
  : Method 수준
-

```
@RestController
Public class ServiceController {

   @RequestMapping(value="/uri/process", method = RequestMethod.POST)
   public ResponseDTO process(@RequestBody RequestDTO requestDTO) { … }

   // 복수 설정
   @RequestMapping(value={"/uri/process", "/ uri/process01"},
                   method = RequestMethod.POST)
   public ResponseDTO process(@RequestBody RequestDTO requestDTO) { … }
}
```

: Class 수준

```
@RestController
@RequestMapping(value="/uri/process")
Public class ServiceController {
    // Uri : /uri/process  -> Http Request Method만 사용
    // RequestMethod.POST, GET, PUT, PATCH, DELETE, TRACE, OPTIONS
@RequestMapping(method = RequestMethod.POST)
   public ResponseDTO process(@RequestBody RequestDTO requestDTO) { … }

   // Uri : /uri/process/processA
   @RequestMapping(value="/processA",
                   method = RequestMethod.POST)
   public ResponseDTO process(@RequestBody RequestDTO requestDTO) { … }

  // 복수 설정
   @RequestMapping(value={"/processA", "/processB"},
                   method = RequestMethod.POST)
   public ResponseDTO process(@RequestBody RequestDTO requestDTO) { … }

}
```

#별첨 1. Spring Boot

01. Spring Boot 란

- Spring Framework에 기반을 으로 확장해 자동 구성을 가능하게 한다.
- 단순히 실행하고 독립형 제품 수준의 스프링 기반 Application을 쉽게 만듦
- JMS, JDBC, JPA등과 같은 하부 구조를 자동으로 구성

02. Application Class 생성

- Main 함수가 있는 Class로 시작점
- @SpringBootApplication
  : @Configuration(스프링 자바 구성 클래스), @ComponenrScan(컴포넌트 탐색), @EnableAutoConfiguration(자동 구성)을 포함

```
@SpringBootApplication
public class ExpFilterInApplication {

   public static void main(String[] args) {
      ApplicationContext  ctx = SpringApplication.run(ExpFilterInApplication.class, args);

      System.out.println("# Beans : " + ctx.getBeanDefinitionCount());

      String[] names = ctx.getBeanDefinitionNames();

      Arrays.sort(names);
      Arrays.asList(names).forEach(System.out::println);
    }
}
```

```
@Component
public class ExpDiRunner implements ApplicationRunner {

   @Override
   public void run(ApplicationArguments args) throws Exception {
      System.out.println("*** ****     **********");
  }

}
```

```
# Beans : 132
GenericFilterBeanFlte
applicationAvailability
applicationTaskExecutor
basicErrorController
beanNameHandlerMapping
beanNameViewResolver
characterEncodingFilter
conventionErrorViewResolver
defaultServletHandlerMapping
defaultViewResolver
dispatcherServlet
```

#별첨 2. Spring Boot 시작

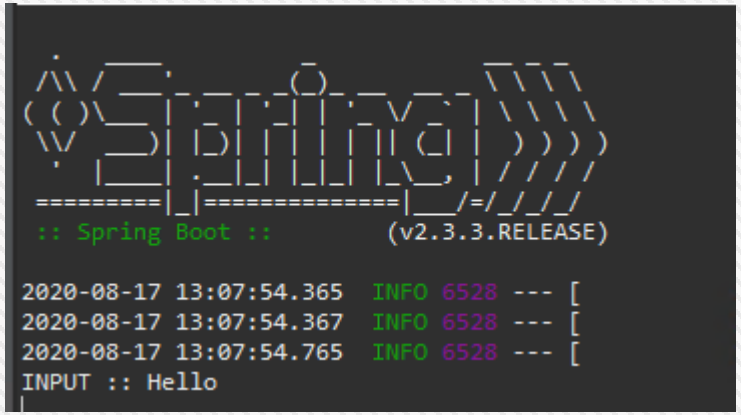
01. ApplicationRunner Interface

- Application이 시작되고 일부 코드를 수행 하고자 할 때 사용

```
@SpringBootApplication
public class ExpRunApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExpRunApplication.class, args);
    }

    @Bean
    public ApplicationRunner StringSVC(ServiceSVC serviceSVC) {
        return args -> {
            System.out.println(serviceSVC.StringSVC("Hello"));
        };
    }

}
```

```
public interface ServiceSVC {
    public String StringSVC(String str);
}
```

```
public interface ServiceSVC {
    public String StringSVC(String str);
}
```



THANKS

ABACUS

www.iabacus.co.kr

Tel. 82-2-2109-5400

Fax. 82-2-6442-5409