

Anyframe Core Plugin



Version 1.6.0

저작권 © 2007-2014 삼성SDS

본 문서의 저작권은 삼성SDS에 있으며 Anyframe 오픈소스 커뮤니티 활동의 목적하에서 자유로운 이용이 가능합니다. 본 문서를 복제, 배포할 경우에는 저작권자를 명시하여 주시기 바라며 본 문서를 변경하실 경우에는 원문과 변경된 내용을 표시하여 주시기 바랍니다. 원문과 변경된 문서에 대한 상업적 용도의 활용은 허용되지 않습니다. 본 문서에 오류가 있다고 판단될 경우 이슈로 등록해 주시면 적절한 조치를 취하도록 하겠습니다.

I. Introduction	1
II. Spring	2
1. IoC(Inversion of Control)	3
1.1. Basic	5
1.1.1. Container와 Bean	5
1.1.2. Container	5
1.1.3. Beans	9
1.1.4. How to refer to Beans	10
1.2. Dependencies	12
1.2.1. Dependency Injection(DI)	12
1.2.2. Bean Property와 생성자 인자	15
1.2.3. depends-on 속성 사용	19
1.2.4. Lazy Instantiation	19
1.2.5. Autowiring	19
1.2.6. Dependency Check	20
1.3. Method Injection	21
1.3.1. Lookup Method Injection	21
1.3.2. Method Replacement	21
1.4. Bean과 Container의 확장	22
1.4.1. Bean Scope	22
1.4.2. Bean Life Cycle	25
1.4.3. Bean 상속	27
1.4.4. Container 확장	28
1.4.5. ApplicationContext 활용	29
1.5. XML 스키마 기반 설정	31
2. Annotation	33
2.1. Bean Management	33
2.1.1. Auto Detecting	34
2.1.2. Using Filters to customize scanning	35
2.1.3. Scope Definition	36
2.2. Dependency Injection	36
2.2.1. @Inject	36
2.2.2. @Autowired	37
2.2.3. @Resource	39
2.2.4. @Qualifier	39
2.2.5. Provider	41
2.2.6. Generic Types as a form of Qualifier	42
2.2.7. Ordered Autowiring in Arrays	42
2.2.8. @Inject / @Autowired / @Resource 비교	43
2.2.9. Inject 시점에 @Lazy 사용	43
2.3. Lifecycle Annotation	46
2.3.1. @PostConstruct	46
2.3.2. @PreDestroy	46
2.3.3. Combining lifecycle mechanisms	47
2.4. Meta-Annotation	47
2.5. Resources	48
3. Bean Definition Profiles	50
3.1. XML Profiles	50
3.1.1. XML 기반의 Profile 적용	51
3.1.2. 중첩된 beans 태그	51
3.2. Profile 활성화	52
3.2.1. Programmatic Profile 활성화	52
3.2.2. 선언적인 Profile 활성화	52
3.2.3. 다중 Profile 활성화	52
3.2.4. @ActiveProfiles	53

3.2.5. ActiveProfilesResolver	53
3.3. @Profile	54
3.3.1. Annotation 기반의 Profile 적용	54
3.4. @Conditional	55
3.4.1. Annotation 기반의 Conditional 적용	55
3.5. Environment Abstraction (Environment 추상화)	56
3.5.1. PropertySources	56
3.5.2. PropertySource 활용	57
3.5.3. 웹 어플리케이션내의 PropertySource 활용	58
4. Java based Configuration	59
4.1. Bean Management	59
4.1.1. Naming	60
4.1.2. Lifecycle Management	60
4.1.3. Scope	60
4.1.4. Dependency Injection	61
4.1.5. Method Injection	63
4.1.6. Spring Expression Language	64
4.1.7. Description Annotation	64
4.1.8. Code Equivalents for Spring's XML namespaces	64
4.2. Combining Java and XML Configuration	65
4.2.1. Combine Java Configuration	66
4.2.2. Combine XML Configuration	66
4.3. Instantiating spring container	67
4.3.1. AnnotationConfigApplicationContext	67
4.3.2. AnnotationConfigWebApplicationContext	67
4.4. Resources	68
5. AOP(Asspect Oriented Programming)	70
5.1. AOP 구성 요소	73
5.1.1. JointPoint	73
5.1.2. Pointcut	73
5.1.3. Advice	75
5.1.4. Weaving 또는 CrossCutting	75
5.1.5. Aspect	76
5.2. Annotation based AOP	77
5.2.1. Configuration	77
5.2.2. @Aspect 정의	77
5.2.3. @Pointcut 정의	77
5.2.4. @Advice 정의	77
5.2.5. Aspect 실행	81
5.3. XML based AOP	82
5.3.1. Aspect 정의	82
5.3.2. Pointcut 정의	82
5.3.3. Advice 정의 및 구현	83
5.4. AspectJ based AOP	87
5.4.1. 시작하기 전에	87
5.4.2. Aspect 정의	88
5.4.3. Pointcut 정의	88
5.4.4. Advice 정의	89
5.5. AOP Examples	91
5.5.1. AOP Example - Logging	91
5.5.2. AOP Example - Exception Transfer	93
5.5.3. AOP Example - Profiler	94
5.5.4. AOP Example - Design Level Assertions	95
5.6. Resources	99
6. SpEL(Spring Expression Language)	100

6.1. Bean Definition using SpEL	100
6.1.1. XML based Bean Definition	100
6.1.2. Annotation based Bean Definition	100
6.2. Expression Evaluation using Spring's Expression Interface	101
6.3. Language Reference	102
6.3.1. Literal Expressions	102
6.3.2. Properties, Arrays, Lists, Maps, Indexers	103
6.3.3. Methods	103
6.3.4. Relational Operators	103
6.3.5. Logical Operators	104
6.3.6. Mathematical Operators	104
6.3.7. Assignment	104
6.3.8. Types	104
6.3.9. Constructors	105
6.3.10. Variables	105
6.3.11. Functions	105
6.3.12. Ternary Operator	106
6.3.13. Elvis Operator	106
6.3.14. Safe Navigation Operator	107
6.3.15. Collection Selection	107
6.3.16. Collection Projection	108
6.3.17. Expression Templating	108
6.3.18. 테스트 데이터 : Genre & Movies	109
6.4. Resources	110
7. DataSource	111
7.1. JDBCDataSource Configuration	111
7.1.1. Samples	111
7.2. DBCPDataSource Configuration	111
7.2.1. Samples	112
7.3. C3PODataSource Configuration	113
7.3.1. Samples	113
7.4. JNDIDataSource Configuration	113
7.4.1. Samples	114
7.4.2. jee schema 를 통한 JNDIDataSource 사용	114
7.5. Test Case	115
7.6. Resources	115
8. Transaction Management	117
8.1. Declarative Transaction Management	121
8.1.1. Annotation을 이용한 Transaction 관리	121
8.1.2. XML 정의를 이용한 Transaction 관리	122
8.1.3. [참고] Propagation Behavior, Isolation Level	124
8.1.4. 테스트 케이스 상세	125
8.2. Programmatic Transaction Management	126
8.2.1. TransactionTemplate을 이용한 Transaction 관리	126
8.2.2. TransactionManager를 직접 이용한 Transaction 관리	128
8.3. Resources	129
III. Spring MVC	131
9. Architecture	132
10. Configuration	133
10.1. web.xml 작성	133
10.1.1. DispatcherServlet 등록	133
10.1.2. Spring MVC 설정 파일 위치 등록	133
10.2. action-servlet.xml 작성	133
10.2.1. Handler Mapping	134
10.2.2. View Resolver	137

10.2.3. Configuration Simplification	138
11. Controller	142
11.1. Configuration	142
11.1.1. Using Filters to customize scanning	142
11.2. 컨트롤러 구현	143
11.2.1. @Controller	144
11.2.2. @RequestMapping	144
11.2.3. @RequestParam	153
11.2.4. @RequestBody	153
11.2.5. @ResponseBody	154
11.2.6. @RestController	154
11.2.7. HttpEntity<?>	155
11.2.8. @ModelAttribute	155
11.2.9. @SessionAttributes	156
11.2.10. @CookieValue	156
11.2.11. @RequestHeader	156
11.3. Double Form Submission 방지	157
12. View	158
12.1. Tag library	158
12.1.1. configuration	158
12.1.2. form	158
12.1.3. input	158
12.1.4. checkbox	158
12.1.5. checkboxes	159
12.1.6. radiobutton	159
12.1.7. radiobuttons	159
12.1.8. password	160
12.1.9. select	160
12.1.10. option	160
12.1.11. options	161
12.1.12. textarea	161
12.1.13. hidden	161
12.1.14. errors	161
12.1.15. sample	161
12.2. Tiles Integration	163
13. Validation	164
13.1. Spring Validator	164
13.1.1. Validator 생성	164
13.1.2. Validator 활용	165
13.1.3. <form:errors> 태그 사용	165
13.2. Spring 3 Validation	165
13.2.1. JSR-303 (Bean Validation) Basic	165
13.2.2. JSR-303 (Bean Validation) Optional	167
13.2.3. Custom Constraints	170
13.2.4. Declarative Validating	171
13.2.5. Programmatic Validating	172
13.3. Resources	173
14. Data Binding and Type Conversion	174
14.1. PropertyEditor	174
14.1.1. Implementing Custom Editor	174
14.1.2. Default PropertyEditors	175
14.1.3. Register Custom Editor	175
14.1.4. PropertyEditor의 단점	177
14.2. Spring 3 Type Conversion	177
14.2.1. Implementing Converter	177

14.2.2. Default Converter	179
14.2.3. Register Converter	179
14.2.4. ConversionService 사용하기	180
14.3. Spring 3 Formatting	181
14.3.1. Implementing Formatter	181
14.3.2. Default Formatter	182
14.3.3. Annotation 기반 Formatting	182
14.3.4. Register Formatter	185
15. File Upload	187
16. Internationalization	188
16.1. 다국어 지원 기능	188
16.1.1. LocaleResolver를 이용한 Locale 변경	189
16.1.2. LocaleChangeInterceptor를 이용한 Locale 변경	189
16.2. LocaleResolver	190
16.2.1. AcceptHeaderLocaleResolver	190
16.2.2. CookieLocaleResolver	190
16.2.3. SessionLocaleResolver	191
17. Exception Handling	193
17.1. 특정 error 페이지로 이동하여 에러 메시지 출력	193
17.2. 에러 페이지에 에러 메시지 출력	194
17.3. Presentation Layer에서 message key를 이용한 locale 변경	194
17.3.1. Business Layer의 BaseException 발생	194
17.3.2. Presentation Layer에서 꺼낸 message key 값에 새로운 Locale로 셋 팅	195
18. Spring Integration	196
18.1. Listener 등록과 Spring 설정 파일 목록 위치 정의	196
18.2. Dependency Injection을 통한 Business Service 호출	196
18.3. Resources	197
IV. Spring MVC Extensions	198
19. Configuration Simplification	199
19.1. <mvc:annotation-driven>	199
19.2. @EnableWebMvcAnyframe	199
19.3. Resources	201
20. Tag library	202
20.1. Page Navigator Tag	202
V. Id Generation	203
VI. Logging	204
21. Configuration	205
21.1. appender	205
21.2. logger	207
21.3. root	207
22. Logging	208
22.1. 기본적인 사용 방법	208
22.2. ResourceBundle을 이용하는 방법	208
23. Resources	210
VII. Test	211
24. Service Code Test	212
24.1. JUnit	212
24.2. Test Code 구현	212
24.3. Test Code 예서의 Meta Annotation 사용	213
25. Controller Code Test	215
25.1. TestCode	215
26. Resources	216
VIII. Message Source	217
27. ReloadableResourceBundleMessageSource	218

28. DatabaseMessageSource	219
28.1. Configuration	219
28.2. Import/Export/Refresh Messages	221
29. AggregatingMessageSource	223
IX. Query Service	225
X. Properties Service	226
30. PropertiesServiceImpl	227
30.1. Samples	227
31. Sample Property File	229
32. Dynamic Reloading	230
33. Resources	231
XI. Servlet 3.0	232
34. Servlet 3 지원 환경	233
35. Ease of Development	234
35.1. Easy configuration through annotations	234
35.1.1. @WebServlet	234
35.1.2. @WebInitParam	235
35.1.3. @WebFilter	236
35.1.4. @WebListener	236
35.1.5. @MultipartConfig	237
35.2. Programmatic Configuration	239
35.3. Resources	241
36. Pluggability	242
36.1. Web Fragment	242
36.2. Resource Sharing	243
36.3. Resources	244
37. Asynchronous Support	245
38. Security Enhancement	246

I.Introduction

Core Plugin은 Anyframe에서 제공하는 모든 Plugin의 기반이 되는 기본 Plugin으로 웹 어플리케이션 개발에 기본적으로 필요한 오픈소스들과 Core Service의 활용 방법을 가이드하기 위한 샘플 코드와 이 오픈소스들을 활용하는데 필요한 참조 라이브러리들로 구성되어 있다.

- 비즈니스 레이어와 프리젠테이션 레이어는 기본적으로 Spring, SpringMVC [<http://www.springsource.org>]를 활용하여 구성되어 있으며 데이터 접근 레이어는 Spring JdbcTemplate을 활용한다.
- 어플리케이션 개발시 공통적으로 요구되는 DB 연동, Logging 등의 기능을 수행하기 위해 오픈소스 Commons DBCP [<http://commons.apache.org/dbcp/>], Log4j [<http://logging.apache.org/log4j/>]를 활용한다.
- 이 외에도 String, Date, Number, Digest, Validation 관련 Utility, 잦은 변경이 요구되는 설정 정보를 외부에서 관리하고 어플리케이션을 통해 해당 정보에 접근할 수 있도록 지원하는 Properties, 프리젠테이션 레이어 개발의 편의 증진을 위한 Custom Tag Library 등을 제공하는 Core 서비스를 활용한다.

Installation

Command 창에서 다음과 같이 명령어를 입력하여 core plugin을 설치한다.

```
mvn anyframe:install -Dname=core
```

installed(mvn anyframe:installed) 혹은 jetty:run(mvn clean jetty:run) command를 이용하여 설치 결과를 확인해볼 수 있다.

Plugin Name	Version Range
datasource	2.0.0 > *
logging	2.0.0 > *
spring	2.0.0 > *

II.Spring

Spring은 객체의 라이프 사이클을 관리하고 객체들간의 의존 관계를 최소화할 수 있는 Lightweight 컨테이너를 제공한다. 다음은 Spring Lightweight 컨테이너의 주요 특징이다.

- POJO 기반 개발 지원

설계 결과물에 컨테이너 의존적인 코드를 추가하지 않아도 순수 POJO 기반으로 어플리케이션 개발이 가능하도록 지원한다. 즉, Lightweight 컨테이너 기반 개발시 프레임워크로 인한 기본 설계와 상세 설계가 이중으로 진행되거나, 개발시 설계 모델과 구현체가 불일치되는 것을 방지할 수 있다.

- Dependency Resolution 지원

어플리케이션 구성 모듈간 의존 관계를 처리하기 위한 방법을 제공한다. 특정 모듈의 코드 내에서 참조할 모듈을 직접적으로 생성하여 참조함으로써 참조 모듈간에 tightly-coupled 되지 않도록 하기 위해, 대부분의 Lightweight 컨테이너들과 마찬가지로 DI(Dependency Injection)을 지원하며, 이외에 DL(Dependency Lookup)도 가능하다.

- Aspect Oriented Programming 지원

AOP는 어플리케이션 전체에 걸쳐 사용되나 쉽게 분리된 모듈로 작성하기 힘든 로깅, 인증, 권한체크, DB 연동, 트랜잭션, 락킹, 에러처리 등과 같은 공통 기능을 재사용 가능하도록 컴포넌트화 할 수 있는 기법이다. AOP에서는 이러한 공통 기능을 Crosscutting Concerns, 해당 어플리케이션이 제공하는 비즈니스 기능을 Core Concerns 라고 지칭한다. 즉, Core Concerns 모듈 내에 필요한 Crosscutting Concerns를 직접 추가하는 대신에 AOP에서는 Weaving이라는 작업을 통해 Core Concerns 모듈의 코드를 직접 건드리지 않고도 Core Concerns 모듈의 사이 사이에 필요한 Crosscutting Concerns 코드가 엮어져 동작되도록 한다. 이를 통해 AOP는 기존의 작성된 코드들을 수정하지 않고도 필요한 Crosscutting Concerns 기능을 효과적으로 적용해 낼 수도 있게 되는 것이다.

- Life-cycle 관리

Lightweight 컨테이너는 정의된 모듈의 Life-cycle을 관리하여 해당 모듈들을 초기화시키고 종료시키는 역할을 수행함으로써 개발자가 비즈니스 로직에 집중하여 개발할 수 있게 된다.

- 신규 기능 추가 용이

XML 또는 Annotation 기반의 설정을 통해서 간단하게 컨테이너 기반 위에 신규 기능을 추가할 수 있도록 지원한다.

여기에서는 Spring Lightweight 컨테이너를 통해 지원되는 주요 기능들에 대해 살펴볼 것이다. 이와 함께 클라이언트 어플리케이션과 원격 어플리케이션에서 제공하는 서비스 간의 의사 소통을 위한 Spring Remoting 기법에 대해서도 알아보자.

1. IoC(Inversion of Control)

Anyframe은 Spring 기반에서 다양한 best-of-breed 오픈 소스를 통합 및 확장하여 구성한 어플리케이션 프레임워크를 포함하고 있다. Anyframe 5.6.0 이후부터는 Spring Framework 4.0을 기반으로 하고 있다.

Spring Framework가 가지는 가장 핵심적인 기능이 IoC이다. IoC 개념은 과거에도 많은 곳에서 사용된 개념이지만 최근 Spring Framework와 같은 Lightweight Container 개념이 등장하면서 많은 개발자들에게 관심의 대상이 되고 있다. IoC 개념은 Spring Framework 뿐만 아니라 컨테이너 기능을 가지는 모든 영역에서 사용되고 있는 개념이므로 반드시 이해할 필요가 있다.

• IoC(Inversion of Control)개념

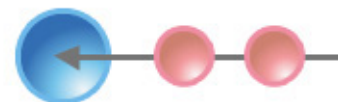
IoC는 Inversion of Control의 약자이다. 우리나라 말로 직역해 보면 "역제어"라고 할 수 있다. 제어의 역전 현상이 무엇인지 살펴본다. 기존에 자바 기반으로 어플리케이션을 개발할 때 자바 객체를 생성하고 서로간의 의존 관계를 연결시키는 작업에 대한 제어권은 보통 개발되는 어플리케이션에 있었다. 그러나 Servlet, EJB 등을 사용하는 경우 Servlet Container, EJB Container에게 제어권이 넘어가서 객체의 생명주기(Life Cycle)를 Container들이 전담하게 된다. 이처럼 IoC에서 이야기하는 제어권의 역전이란 객체의 생성에서부터 생명주기의 관리까지 모든 객체에 대한 제어권이 바뀌었다는 것을 의미한다. Spring Framework도 객체에 대한 생성 및 생명주기를 관리할 수 있는 기능을 제공하고 있다. 즉, IoC Container 기능을 제공하고 있다.

Inversion of Control(이하 IoC)이란?

- Component dependency resolution, configuration 및 lifecycle을 해결하기 위한 Design Pattern
- DIP(Dependency Inversion Principle) 또는 Hollywood Principle (Don't call us we will call you)라는 용어로도 사용
- 특정 작업을 수행하기 위해 필요한 다른 컴포넌트들을 직접 생성하거나 획득하기 보다는 이러한 의존성들을 외부에 정의하고 컨테이너에 의해 공급받는 방법으로 동작



< IoC가 아닌 경우 >

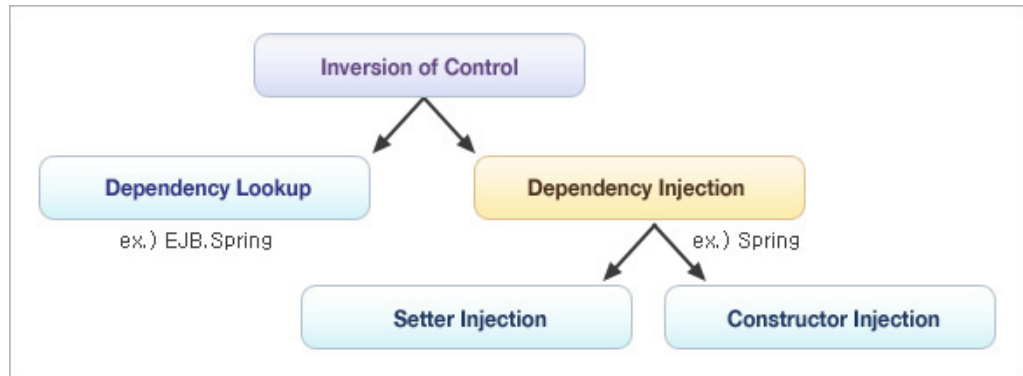


< IoC인 경우 >

이러한 IoC는 다음과 같은 장점을 가지고 있다.

- 클래스 / 컴포넌트의 재사용성 증가
- 단위 테스트 용이
- Assemble과 configure를 통한 시스템 구축 용이
- IoC와 Dependency Injection간의 관계

Spring Framework의 가장 큰 장점으로 IoC Container 기능이 부각되어 있으나, IoC 기능은 Spring Framework가 탄생하기 훨씬 이전부터 사용되던 개념이었다. 그러므로 "IoC 기능을 Spring Framework의 장점이라고 이야기하는 것은 적합하지 않다."고 반론을 제기하면서 "새로운 개념을 사용하는 것이 적합하다."고 주장한 사람이 Martin Fowler이다. Lightweight 컨테이너들이 이야기하는 IoC를 Dependency Injection이라는 용어로 사용하는 것이 더 적합하다고 이야기하고 있다. Martin Fowler의 이 같은 구분 이후 IoC 개념을 개발자들마다 다양한 방식으로 분류하고 있으나 다음 그림과 같이 IoC와 Dependency Injection 간의 관계를 분류하는 것이 일반적이다.



• Dependency Lookup

저장소에 저장되어 있는 Bean에 접근하기 위하여 Container에서 제공하는 API를 이용하여 사용하고자 하는 Bean을 Lookup 하는 것을 말한다. 따라서, Bean을 개발자가 직접 Lookup하여 사용함으로써 Container에서 제공하는 API와 의존관계 발생하게 된다.

• 객체 관리 저장소(Repository)

모든 IoC Container는 각 Container에서 관리해야 하는 객체들을 관리하기 위한 별도의 저장소(Repository)를 가진다. Servlet Container는 web.xml에서 Servlet을 관리하고 있으며, EJB Container는 ejb-jar.xml에 설정되어 있는 정보들이 JNDI 저장소에 저장되어 관리되고 있다. 이처럼 Spring Framework도 POJO들을 관리하기 위하여 별도의 저장소로 XML 파일을 가지게 된다.

• Dependency Lookup 예시

구현 클래스는 다음과 같이 작성한다.

```

public class IoServiceImpl1 implements IoService1,
    ApplicationContextAware {
    public void setApplicationContext (ApplicationContext context) {
        IoService2 iocService2 = (IoService2)context.getBean("IoService2");
    }
}
  
```

속성 정의 파일은 다음과 같이 작성한다.

```

<bean id="IoService1" class="...IoServiceImpl1">
    중략...
</bean>
<bean id="IoService2" class="...IoServiceImpl2">
    중략...
</bean>
  
```

• Dependency Injection (DI)

각 클래스 사이의 의존관계를 빈 설정(Beans Definition)정보를 바탕으로 컨테이너가 자동적으로 연결해주는 것을 말한다. 컨테이너가 의존관계를 자동적으로 연결시켜주기 때문에 개발자들이 컨테이너 API를 이용하여 의존관계에 관여할 필요가 없게 되므로 컨테이너 API에 종속되는 것을 줄일 수 있다. 개발자들은 단지 빈 설정파일(저장소 관리 파일)에서 의존관계가 필요하다는 정보를 추가하기만 하면 된다. 또한 Dependency Injection은 Setter Injection과 Constructor Injection 형태로 구분한다.

• Dependency Injection 예시

구현 클래스는 다음과 같이 작성한다.

```
public class IoServiceImpl implements IoService {
    public void setDependencyBean(DepBean dependencyBean) {
        this.dependencyBean = dependencyBean;
    }
    중략...
}
```

속성 정의 파일은 다음과 같이 작성한다.

```
<bean id="IoService" class="....IoServiceImpl">
    <property name="dependencyBean" ref="depBean"/>
</bean>
```

• Dependency Lookup과 Dependency Injection의 차이점

Bean을 개발자가 직접 Lookup하여 사용하는 것을 Dependency Lookup이라고 하고, Dependency Injection은 이와 달리 각 계층 사이, 각 클래스 사이에 필요로 하는 의존관계가 있다면 이 같은 의존관계를 Container가 자동적으로 연결시켜주는 것을 말한다. Dependency Lookup을 사용할 경우 Bean을 Lookup하기 위하여 Container에서 제공하는 API와 의존관계가 발생한다. 이처럼 Container API와 많은 의존관계를 가지면 가질수록 어플리케이션이 Container에 대하여 가지는 종속성은 증가할 수 밖에 없다. 따라서 가능한 Dependency Lookup을 사용하지 않는 것이 Container와의 종속성을 줄일 수 있게 된다. Container와의 종속성을 줄이기 위한 방법으로는 이후에 다루게 될 Dependency Injection을 통하여 가능하게 된다.

1.1.Basic

Spring Framework는 기본적으로 어플리케이션의 비즈니스 서비스를 구동시키고 관리하는 Spring Container와 이러한 Container에 의해 관리되는 Bean으로 구성된다. Bean은 Container를 통해서 인스턴스화되는 객체이며 Container에 의해 다른 Bean들과 Wiring(연결)되고 관리된다.

1.1.1.Container와 Bean

Bean은 Spring Framework에서 어플리케이션의 중요 부분을 형성하고 Spring IoC Container에 의해 관리된다.

- Bean 설정, 생성, Life Cycle 관리
- Bean Wiring(연결) - Bean들과 각각에 대한 Dependency 관계는 Spring IoC Container에 의해 사용되는 설정 메타데이터로 반영



1.1.2.Container

Spring IoC Container는 다음 두 가지 유형의 Container를 제공한다.

- **BeanFactory**

설 명
Bean의 생성과 소멸 담당
Bean 생성 시 필요한 속성 설정
Bean의 Life Cycle에 관련된 메소드 호출
다수의 BeanFactory 인터페이스 구현 클래스를 제공하며 이중 가장 유용한 것은 XmlBeanFactory임

- **ApplicationContext**

설 명
BeanFactory의 모든 기능 제공
ResourceBundle 파일을 이용한 국제화(I18N) 지원
다양한 Resource 로딩 방법 제공
이벤트 핸들링
Context 시작 시 모든 Singleton Bean을 미리 로딩(preloading) 시킴-> 초기에 설정 및 환경에 대한 에러 발견 가능함
다수의 ApplicationContext 구현 클래스 제공(XmlWebApplicationContext, FileSystemXmlApplicationContext, ClassPathXmlApplicationContext)

org.springframework.beans 와 org.springframework.context 패키지가 Spring Framework의 IoC Container를 위한 기본을 제공한다. BeanFactory는 객체를 관리하는 고급 설정 기법을 제공하고 ApplicationContext는 Spring의 AOP기능, 메시지 자원 핸들링, 이벤트 위임, 웹 어플리케이션에서 사용하기 위한 WebApplicationContext와 같은 특정 ApplicationContext 통합과 같은 기능을 추가 제공한다. 즉, BeanFactory가 설정 프레임워크와 기본 기능을 제공하는 반면 ApplicationContext는 BeanFactory의 모든 기능 뿐 아니라 전사적 중심의 기능이 추가되어 있다. ApplicationContext가 제공하는 부가 기능과는 별개로, ApplicationContext와 BeanFactory의 또 다른 차이점은 Singleton Bean을 로딩하는 방법에 있다. BeanFactory는 getBean() 메소드가 호출될 때까지 Bean의 생성을 미룬다. 즉 BeanFactory는 모든 Bean을 늦게 로딩(Lazy loading)한다. ApplicationContext는 Context를 시작시킬 때 모든 Singleton Bean을 미리 로딩함으로써, 그 Bean이 필요할 때 즉시 사용될 수 있도록 보장해준다. 즉, 어플리케이션 동작 시 Bean이 생성되기를 기다릴 필요가 없게 된다.

1.1.2.1.BeanFactory

Bean을 포함하고 관리하는 책임을 지는 Spring IoC Container의 실제 표현이다.가장 공통적으로 사용되는 BeanFactory의 구현체인 XmlBeanFactory 클래스는 XML 형태로 어플리케이션과 객체간의 참조 관계를 조합하는 객체를 정의함으로써 XML 설정 메타데이터를 기반으로 완전히 설정된 시스템이나 어플리케이션을 생성한다. 또한 아래의 예와 같이 XmlBeanFactory는 XML 파일에 기술되어 있는 정의를 바탕으로 Bean을 Loading해준다. (생성자에org.springframework.core.io.Resource타입의 객체 넘겨줌)

```
BeanFactory factory = new XmlBeanFactory(new FileInputStream("beans.xml"));
```

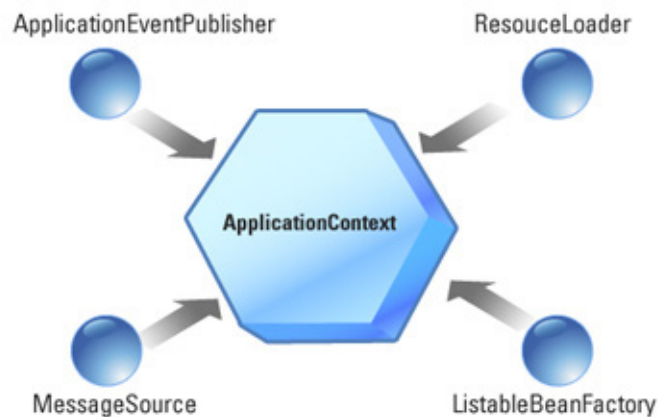
org.springframework.beans.factory.BeanFactory인터페이스에 관한 API는 여기 [http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/beans/factory/BeanFactory.html]를 참고한다.

Resource Implementation	Purpose
org.springframework.core.io.ByteArrayResource	Defines a resource whose content is given by an array of bytes
org.springframework.core.io.ClassPathResource	Defines a resource that is to be retrieved from the classpath
org.springframework.core.io.DescriptiveResource	

Resource Implementation	Purpose
	Defines a resource that holds a resource description but no actual readable resource
org.springframework.core.io.FileSystemResource	Defines a resource that is to be retrieved from the file system
org.springframework.core.io.InputStreamResource	Defines a resource that is to be retrieved from an input stream
org.springframework.web.portlet.context.PortletContextResource	Defines a resource that is available in a portlet context
org.springframework.web.context.support.ServletContextResource	Defines a resource that is available in a servlet context
org.springframework.core.io.UrlResource	Defines a resource that is to be retrieved from a given URL

1.1.2.2.ApplicationContext

다음은 org.springframework.context.ApplicationContext 인터페이스의 대략적인 구조이다.



자주 사용되는 ApplicationContext의 구현 클래스는 아래와 같다.

- XmlWebApplicationContext - 웹 기반의 Spring 어플리케이션을 작성할 때 내부적으로 사용
- FileSystemXmlApplicationContext - 파일 시스템에 위치한 XML 설정 파일을 읽어들이는 ApplicationContext
- ClassPathXmlApplicationContext - 클래스패스에 위치한 XML 설정 파일을 읽어들이는 ApplicationContext

ApplicationContext 구현 클래스를 아래와 같이 사용할 수 있다.

```
ApplicationContext context = new FileSystemXmlApplicationContext("c:/beans.xml");
```

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

1.1.2.3.설정 메타데이터

Container에 의해 "인스턴스화, 설정, 그리고 조합[어플리케이션내 객체를]"하기 위한 설정 방법에 대해 알아 보기로 하자. 대부분은 간단하고 직관적인 XML 형태로 제공되며 XML 기반의 설정 메타데이터를 사용하여 Bean을 정의하도록 한다. 다음은 XML 기반의 설정 메타데이터의 기본 구조 예제이다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">
  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>
  <!-- more bean definitions go here -->
</beans>
```

XML 기반의 메타데이터 정의는 설정 메타데이터의 가장 많이 사용되는 형태이다. XML 외에 Java Properties 파일을 이용하거나 프로그램으로 처리(Spring의 Public API를 사용하여)함으로써, 설정 메타데이터를 제공할 수 있다. Spring IoC Container 자체는 설정 메타데이터의 형태로부터 분리될 수 있기 때문이다.

1.1.2.4.Spring IoC Container 인스턴스화 시키는 예제

1. BeanFactory 사용한 예제

```
Resource resource = new FileSystemResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

```
ClassPathResource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

2. ApplicationContext 사용한 예제

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String ("beans.xml"));
// of course, an ApplicationContext is just a BeanFactory
BeanFactory factory = (BeanFactory) context;
```

1.1.2.5.XML 기반 설정 메타데이터 조합

XML 기반 설정 메타데이터는 다중 XML파일로 분리하여 정의할 수 있다. 여기서 주의할 점은 <import>를 <bean> 이전에 두어야만 하는 것이다.

```
<beans>
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>

  <bean id="bean1" class="..." />
  <bean id="bean2" class="..." />
</beans>
```

위의 예제에서 외부 Bean정의는 3개의 파일(services.xml, messageSource.xml, 과 themeSource.xml)로부터 로드된다. 모든 위치 경로는 import를 수행하는 XML 파일에 상대적이다. 그래서 이 경우에 messageSource.xml 과 themeSource.xml이 import 대상 XML 파일의 위치 아래의 resources 에 두어야 하는 반면에 services.xml은 import를 수행하는 파일과 같은 디렉토리나 클래스패스 경로 내에 두어야만 한다. 이 예제처럼 /는 실제로 무시된다. import된 파일의 내용은 <beans>를 가장상위 레벨에 포함하는 스키마나 DTD에 따라 완전히 유효한 XML Bean 정의 파일 이어야만 한다.

1.1.3.Beans

1.1.3.1.Bean

Spring IoC Container에 의해 관리되는 객체로 Container에 제공된 설정 메타데이터 내 정의(대개 XML <bean> 형태로)에 의해 생성되며 실제로 아래 표로 나타난 주요 메타데이터 정보를 포함하는 BeanDefinition 객체로 표현한다.

주요 메타데이터속성	설명
id	Bean의 구분을 위한 정보로 해당 bean에 접근하기 위한 Key임
class	정의된 Bean의 실제 구현클래스로 항상 full name으로 작성
scope	정의된 Bean의 인스턴스 생성 유형 정의. singleton, prototype, request, session, globalSession 중 선택. Default는 singleton이며, 보다 자세한 Bean Scope에 대해서는 본 매뉴얼의 Extensions Bean Scope 을 참고하도록 한다.
init-method	해당 bean이 초기화된 후 context에 저장되기 전 호출되는 초기화 메소드 정의
destroy-method	해당 bean 제거 시 호출되는 메소드 정의
factory-method	해당 bean 생성 시 생성자를 사용하지 않고 특정 factory method를 호출하여 생성 시 정의
lazy-init	true/false 값을 가지며 해당 bean이 호출되기 전에 초기화 시킬지 여부를 결정함. Default는 false이며 true인 경우, 해당 bean이 호출되는 시점에 초기화됨

1.1.3.2.Bean 명명하기

Bean 정의 시 Bean들을 구분하기 위해 'id' 혹은 'name' 속성을 사용하는데 'id'를 사용하는 경우, 하나의 Bean은 Container내에서 Unique한 id를 가지도록 한다. 일반적으로 Bean을 명명할때 인스턴스 필드명에 대한 표준 Java 규칙을 사용한다. Bean 이름은 소문자로 시작하고 camel-cased(첫 번째 단어는 소문자로 시작하고 두 번째 단어는 대문자로 시작)된다. 이러한 이름의 예제는 'genreService', 'movieDao', 'movieFinderController' 등이다. Bean을 명명하는 일관적인 방법을 적용하는 것은 설정을 좀 더 읽기 쉽고 이해하기 쉽도록 만들어준다. 이러한 명명표준을 적용하는 것은 어려운 일이 아니다. Spring AOP를 사용한다면 특정 Bean 이름과 관련된 Bean의 세트에 advice를 적용할 때 용이해질 수 있다.

1.1.3.3.Bean 인스턴스화

• 생성자를 이용한 인스턴스화

특정 인터페이스를 구현하거나 특정 형태로 코딩 할 필요가 없다.

```
<bean id="sampleBean" class="sample.SampleBean"/>
<bean name="anotherSample" class="sample.SampleBeanTwo"/>
```

• static factory 메소드를 사용한 인스턴스화

Bean 객체가 factory 메소드를 호출하여 생성되는 것으로, 반환 객체의 타입을 명시하지 않고 factory 메소드를 포함하는 클래스를 정의하고 있음에 주의한다. 아래 예제에서 createInstance() 메소드는 static 메소드이어야 한다.

```
<bean id="sampleBean" class="sample.SampleBean2"
factory-method="createInstance"/>
```


- 인스턴스 **factory** 메소드를 사용한 인스턴스화

'class' 속성을 정의하지 않고 'factory-bean' 속성에 factory메소드를 포함하는 Bean을 정의한다.

```
<!-- the factory bean, which contains a method
called createInstance() --><bean id="myFactoryBean" class="..." />
<!-- the bean to be created via the factory bean -->
<bean id="sampleBean"
      factory-bean="myFactoryBean"
      factory-method="createInstance" />
      종 략...
```

1.1.4.How to refer to Beans

비즈니스 레이어와 프리젠테이션 레이어에서 Spring Bean에 접근하는 방법에는 여러 가지가 있다.

1.1.4.1.비즈니스 레이어

비즈니스 레이어에서 사용하고자 하는 Spring Bean에 접근하는 방법은 크게 2가지 형태로 구분할 수 있다. Dependency Lookup과 Dependency Injection 방식이 그것이다.

- **Dependency Lookup**

저장소에 저장되어 있는 Bean에 접근하기 위하여 사용하고자 하는 Bean을 Lookup 한다. 이때 Bean을 개발자가 직접 Lookup하여 사용함으로써 Container에서 제공하는 API와 의존관계가 발생한다. Spring IoC 컨테이너 Dependency Lookup에 대한 자세한 사항은 본 매뉴얼의 IoC 를 참고한다.

구현 클래스는 다음과 같이 작성한다.

```
public class IoServiceImpl1 implements IoService1,
    ApplicationContextAware {
    public void setApplicationContext (ApplicationContext context) {
        IoService2 iocService2 = (IoService2)context.getBean("IoService2");
    }
}
```

속성 정의 파일은 다음과 같이 작성한다.

```
<bean id="IoService1" class="...IoServiceImpl1">
    종 략...
</bean>
<bean id="IoService2" class="...IoServiceImpl2">
    종 략...
</bean>
```

- **Dependency Injection**

각 클래스 사이에 필요로 하는 의존 관계가 있는 경우, 의존관계를 Container가 자동적으로 연결시켜 줌으로써 Container에서 제공하는 API와 의존관계가 없다. Spring IoC 컨테이너 Dependency Injection에 대한 자세한 사항은 본 매뉴얼의 Dependencies를 참고한다.

구현 클래스는 다음과 같이 작성한다. 이 예제에서는 Setter Injection 방식을 보여주고 있다.

```
public class IoServiceImpl implements IoService {
    public void setDependencyBean(DepBean dependencyBean) {
        this.dependencyBean = dependencyBean;
    }
    종 략...
}
```

속성 정의 파일은 다음과 같이 작성한다.

```
<bean id="IoCService" class="...IoCServiceImpl">
  <property name="dependencyBean" ref="depBean"/>
</bean>
```

1.1.4.2. 프리젠테이션 레이어

프리젠테이션 레이어에서 Spring Bean에 접근하는 방법은 비즈니스 레이어와 마찬가지로 Dependency Lookup과 Dependency Injection 방식 2가지 중 선택할 수 있는데 이때 사용하는 Web Framework이 무엇인지에 따라 사용 가능한 방식이 제한될 수 있으므로 주의하도록 한다. Web Framework 사용과 관련된 설정 방법은 Spring MVC를 참조하도록 한다.

• Dependency Lookup (Struts)

Web Framework으로 Struts를 사용하는 경우, Struts Action 내에서 Spring의 Web ApplicationContext를 얻어내어 Spring Bean을 Lookup하도록 한다. Spring에서 제공하는 ActionSupport 클래스의 getWebApplicationContext() 메소드를 이용하여 ApplicationContext를 얻는다.

Action 클래스는 다음과 같이 작성한다. 이 예제는 Anyframe 을 이용하여 작성된 코드로 Anyframe 의 DefaultActionSupport를 상속한 UpdateMovieAction 클래스의 일부이다. movieService Bean을 사용하고 있으며 이때 Bean의 id 값이 Action 클래스에 명시되어야 함에 유의하도록 한다.

```
public class UpdateMovieAction extends DefaultActionSupport {
    public ActionForward process(ActionMapping mapping, ActionForm form,
        HttpServletRequest req, HttpServletResponse res) throws Exception {
        ApplicationContext ctx = getWebApplicationContext();
        MovieService movieService = (MovieServiceImpl) ctx.getBean ("movieServiceImpl");
        종략...
    }
}
```

속성 정의 파일은 다음과 같이 작성한다.

```
<bean id="movieServiceImpl"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    종략...
</bean>
```

• Dependency Injection (Spring MVC)

Web Framework으로 Spring MVC를 사용하는 경우, Controller 클래스 내에서 Dependency Injection 방식을 이용하여 Spring Bean을 참조할 수 있다.

Controller 클래스는 다음과 같이 작성한다. 이 예제는 Anyframe 을 이용하여 작성된 MovieController 클래스의 일부이다. movieService Bean을 사용하고 있으며 이때 Bean의 id 값이 Spring MVC 속성 정의 파일에서 정의되고 있다.

```
public class MovieController {
    private MovieService movieService;;

    public void setMovieService(MovieService movieService) {
        this.movieService = movieService;
    }
    종략...

    public ModelAndView list(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Movie movie = new Movie();
```

```

        bind(request, movie);
        Page resultPage = movieService.getPagingList(movie);
        중략...
    }
}

```

속성 정의 파일은 다음과 같이 작성한다.

```

<bean name="/coreMovie.do"
      class="org.anyframe.plugin.core.moviefinder.web.MovieController">
    <property name="movieService" ref="coreMovieService"/>
    <property name="genreService" ref="coreGenreService"/>
</bean>

```

• Dependency Lookup (Spring MVC)

Web Framework으로 Spring MVC를 사용하는 경우, Controller 클래스가 아닌 일반 클래스에서 Dependency Lookup 방식으로 Spring Bean을 참조할 수 있다. 웹에서 Spring 설정 파일을 읽어들이고, WebApplicationContext를 생성하고 이것을 해당 웹 어플리케이션의 ServletContext에 저장하므로 ServletContext에 접근 가능하다면 일반 클래스에서도 WebApplicationContext를 얻어낼 수 있게 된다.

일반 클래스에서 다음과 같이 작성한다.

```

WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(servletContext);
MovieService movieService = (MovieServiceImpl)ctx.getBean("movieServiceImpl");
중략...

```

속성 정의 파일은 다음과 같이 작성한다.

```

<bean id="movieServiceImpl"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    중략...
</bean>

```

1.2.Dependencies

전형적인 기업용 어플리케이션은 한 개의 객체(또는 Spring내 Bean)로 만들어지지 않는다는 가장 간단한 어플리케이션조차도 함께 작동하는 소량의 객체를 가진다는 것을 의심할 필요가 없을 것이다. 이 장에서는 독립적인 많은 수의 Bean들이 객체가 몇 가지 목표(대개 최종사용자가 원하는 것을 수행하는 어플리케이션)를 달성하기 위해 함께 작동하는 방법에 대해 알아보기로 한다.

1.2.1.Dependency Injection(DI)

각 클래스 사이의 의존관계를 빈 설정(Been Definition)정보를 바탕으로 컨테이너가 자동적으로 연결해주는 것을 말한다. 컨테이너가 의존관계를 자동적으로 연결시켜주기 때문에 개발자들이 컨테이너 API를 이용하여 의존관계에 관여할 필요가 없게 되어 컨테이너 API에 종속되는 것을 줄일 수 있고 개발자들은 단지 Bean 설정파일(저장소 관리 파일)에서 의존 관계가 필요하다는 정보를 추가하기만 하면 된다. 이는 Setter Injection과 Constructor Injection 형태로 구분한다.

1.2.1.1.Setter Injection

setter 메소드 구현을 통해 초기화 시 Container로부터 의존 관계에 놓인 특정 리소스를 할당받는 방법으로 인자가 없는 생성자나 인자가 없는 static factory 메소드가 Bean을 인스턴스화하기 위해 호출된 후 Bean의 setter 메소드를 호출하여 실제화된다. 다음은 구현 클래스인 MovieServiceImpl.java 의 Setter Injection 부분이다.

```
public class MovieServiceImpl implements MovieService {
    public void setMovieDao(MovieDao movieDao) {
        this.movieDao = movieDao;
    }
    중략...
}
```

다음은 Setter Injection 속성 정의 파일인 context-core.xml 의 일부이다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    <property name="movieDao" ref="coreMovieDao" />
</bean>

<bean id="coreMovieDao"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao">
</bean>
```

1.2.1.2.Constructor Injection

Constructor 구현을 통해 초기화 시 Container로부터 의존 관계에 놓인 특정 리소스를 할당받는 방법으로 각각의 협력자를 표시하는 다수의 인자를 가진 생성자를 호출하여 실제화된다. 추가적으로, Bean을 생성하기 위한 특정 인자를 가진 static factory 메소드를 호출하는 것은 대부분 동등하게 간주될 수 있다. 다음은 구현 클래스인 MovieServiceImpl.java 의 Constructor Injection 부분이다.

```
public class MovieServiceImpl implements MovieService {
    MovieDao movieDao;
    public MovieServiceImpl(MovieDao movieDao) {
        super(movieDao);
        this.movieDao = movieDao;
    }
    중략...
}
```

다음은 Constructor Injection 속성정의 파일인 context-core.xml 의 일부이다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    <constructor-arg ref="coreMovieDao"/>
</bean>

<bean id="coreMovieDao"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao">
    중략...
</bean>
```

type 속성 정의를 이용하면, Constructor의 argument에 대한 클래스 타입을 명시적으로 정의할 수도 있다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    <constructor-arg type="org.anyframe.plugin.core.moviefinder.service.BeanA" ref="beanA"/>
    <constructor-arg type="org.anyframe.plugin.core.moviefinder.service.BeanB" ref="beanB"/>
</bean>
```

Constructor의 argument 개수가 2개 이상이고, 동일한 클래스 타입의 argument가 존재할 경우 모호함을 없애기 위해, index 속성 정의를 통해 argument의 순서대로 할당할 값을 정의할 수 있다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
```

```
<constructor-arg index="0" ref="beanA" />
<constructor-arg index="1" ref="beanB" />
</bean>
```

1.2.1.3.Setter Injection vs. Constructor Injection

Setter Injection 장점	Constructor Injection 장점
- 생성자 Parameter 목록이 길어 지는 것 방지	- 강한 의존성 계약 강제
- 생성자의 수가 많아 지는 것 방지	- Setter 메소드 과다 사용 억제
- Circular dependencies 방지	- 불필요한 Setter 메소드를 제거함으로써 실수로 속성 값을 변경하는 일을 사전에 방지

- Circular dependencies

Constructor Injection 사용 시 주의해야 한다. 다음과 같이 두 개의 서로 다른 Bean이 생성자 Argument 로 서로의 Bean을 참조하는 경우가 그 예이다.

```
<bean id="beanFirst" class="test.BeanFirst">
  <constructor-arg ref="beanSecond" />
</bean>

<bean id="beanSecond" class="test.BeanSecond">
  <constructor-arg ref="beanFirst" />
</bean>
```

1.2.1.4.생성자 인자 분석

생성자 인자 분석 시 사용되는 방법에는 타입 대응과 인덱스가 있다.

- 생성자의 인자 타입 대응(match)

'type' 속성을 사용하여 생성자의 인자 타입을 명확하게 명시함으로써 간단한 타입으로의 타입 매치를 사용할 수 있다.

```
<bean id="sampleBean" class="sample.SampleBean">
  <constructor-arg type="int"><value>7500000</value></constructor-arg>
  <constructor-arg type="java.lang.String"><value>42</value></constructor-arg>
</bean>
```

- 생성자의 인자 인덱스

생성자의 인자는 index 속성을 사용하여 명확하게 명시된 인덱스를 가질 수 있다. 또한 인덱스를 명시하는 것은 생성자의 인자들이 같은 타입을 가질 경우 발생하는 모호함의 문제도 해결한다. (인덱스는 0 부터 시작된다 는 것에 주의하여야 한다.)

```
<bean id="sampleBean" class="sample.SampleBean">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</bean>
```

1.2.1.5.c-namespace 기반의 XML 단축표현

Spring 3.1 버전부터 소개된 c-namespace를 이용하면, 중첩하여 표현된 복잡한 constructor-arg 요소를 축약된 형태의 생성자 인자 속성으로 간단히 나타낼 수 있다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:c="http://www.springframework.org/schema/c"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="bar" class="x.y.Bar"/>
<bean id="baz" class="x.y.Baz"/>

<-- 'traditional' declaration -->
<bean id="foo" class="x.y.Foo">
  <constructor-arg ref="bar"/>
  <constructor-arg ref="baz"/>
  <constructor-arg value="foo@bar.com"/>
</bean>

<-- 'c-namespace' declaration -->
<bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com">

</beans>

```

간혹 생성자의 인자값을 알 수 없는 경우 (일반적으로 디버깅 정보없이 컴파일된 bytecode의 경우), 다음과 같이 대안으로 인자의 순번을 이용할 수 있다.

```

<-- 'c-namespace' index declaration -->
<bean id="foo" class="x.y.Foo" c:_0-ref="bar" c:_1-ref="baz">

```



참고

XML 문법의 제약사항으로 인하여, XML의 속성명은 숫자로 시작할 수 없다. 따라서 순번을 이용하는 경우에는 _(언더바) 문자를 XML 속성명 앞에 표기하여야 한다.

실제 상황에서, 생성자 분석 메커니즘은 굳이 순번을 이용할 필요가 없을 정도로 상당히 효율적이다. 따라서 설정시 생성자의 순번 대신 인자명을 기술하는 방식을 권장한다.

1.2.2.Bean Property와 생성자 인자

Bean Property와 생성자의 인자는 다른 관리 Bean(협력자), 또는 인라인으로 정의된 값을 참조할 수 있다. Spring의 XML-기반의 설정 메타데이터는 이러한 목적을 위한 <property> 와 <constructor-arg> 내에 서 많은 수의 하위 태그를 지원한다.

• Primitive Type - 순수값 지원

<value>는 사람이 읽을 수 있는 문자열 표현처럼 Property나 생성자의 인자를 명시한다.

```

<bean id="myDataSource" destroy-method="close">
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
</bean>

```

• <ref> 요소

다른 bean에 대한 참조인 <ref>는 <constructor-arg> 또는 <property> 내부에 허용되는 마지막 요소이다. 이것은 Container에 의해 관리되는 다른 Bean을 참조하기 위해 Property의 값을 셋팅하는데 사용된다. 모든 참조는 궁극적으로 다른 객체에 대한 참조이지만 다른 객체의 id/name을 명시하는 방법은 3가지가 있다. <ref>의 bean 속성을 사용하여 대상 bean을 명시하는 것이 가장 일반적인 형태이고 같은 Container(같은 XML파일이든 아니든)나 부모 Container 내에서 어떠한 Bean에 대한 참조를 생성하는 것을 허용할 것이다. 'bean' 속성의 값은 대상 bean의 'id' 속성이나 'name' 속성의 값 중 하나가 될 것이다.

- 타 Bean 참조

```
<!-- 'bean' 속성 값은 타 Bean의 'id' 속성 혹은 'name' 속성이자. -->
<ref bean="someBean"/>
```

```
<!-- 'local' 속성 값은 동일 XML 파일 내 타 Bean의 'id' 속성이자. -->
<ref local="someBean"/>
```

- parent context에 존재하는 타 Bean 참조(parent 속성 사용)

```
<!-- in the parent context -->
<bean id="accountService"
      class="com.foo.SimpleAccountService">
    <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="movieService" class="com.foo.SimpleMovieService">
    <ref parent="accountService"/>
</bean>
```

- inner Bean

<property> 나 <constructor-arg> 내부의 <bean> 은 inner bean이라 불리는 것을 정의하기 위해 사용된다. inner bean 정의시 언급된 id나 name, scope값은 Container에 의해 무시되기 때문에 id나 name 값을 명시하지 않는 것이 가장 좋다. inner bean은 언제나 익명이고 prototype 형태로 동작한다.

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target inline -->
    <property name="target">
        <bean class="com.mycompany.Person">
            <!-- this is the inner bean -->
            <property name="name" value="Fiona Apple"/>
            <property name="age" value="25"/>
        </bean>
    </property>
</bean>
```

- Collection

<list> , <set> , <map>과 <props>은 Java Collection의 List, Set, Map and Properties의 타입으로 매핑된다. 또한 객체 Array 타입의 경우에도 콤마(,)를 이용하여 값을 설정할 수 있다(ex. String[]).

```
<bean id="moreComplexObject" class="sample.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@somecompany.org</prop>
        </props>
    </property>

    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
```

```

<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
    <entry>
      <key>
        <value>entry key</value>
      </key>
      <value>entry value</value>
    </entry>
  </map>
</property>

<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
  <set>
    <value>just some string</value>
    <ref bean="myDataSource" />
  </set>
</property>

<!-- results in a setSomeArray(String[]) call -->
<property name="someArray" value="str1,str2,str3,str4"/>
</bean>

```

• Collection 병합

부모역할을 하는 <list>, <map>, <set> 또는 <props>를 정의하고 이를 상속받는 <list>, <map>, <set> 또는 <props>를 정의하는 것이 가능하다. 예를 들면, 자식 collection의 값은 부모 collection내 명시된 값과 자식 collection내 명시된 값을 병합하여 얻어진다.

예제설명) child Bean의 adminEmails Property의 <props>에서 **merge=true**속성을 사용하면, child Bean이 Container에 의해 실질적으로 분석되고 인스턴스화 될때, 부모의 adminEmails collection과 자식의 adminEmails collection이 병합된 형태의 adminEmails collection 을 가지게 된다. 이 병합 행위는 <list>, <map>, 그리고 <set> collection 타입에 유사하게 적용된다. 단, <list>의 경우, 이 의미는 List collection 타입과 관련된다. 이를테면, value의 ordered collection의 개념은 유지관리된다. 부모값은 모든 자식 목록의 값에 선행한다. Map, Set, Properties collection 타입의 경우, Container에 의해 내부적으로 사용되는 Map, Set 그리고 Properties 객체 타입에 관련된 collection 타입의 영향을 받는다.

```

<beans>
  <bean id="parent" abstract="true" class="sample.ComplexObject">
    <property name="adminEmails">
      <props>
        <prop key="administrator">administrator@somecompany.com</prop>
        <prop key="support">support@somecompany.com</prop>
      </props>
    </property>
  </bean>
  <bean id="child" parent="parent">
    <property name="adminEmails">
      <!-- the merge is specified on the *child* collection definition -->
      <props merge="true">
        <prop key="sales">sales@somecompany.com</prop>
        <prop key="support">support@somecompany.co.uk</prop>
      </props>
    </property>
  </bean>
</beans>

```

위 설정 결과 adminEmails Collection은 다음과 같이 구성된다.

administrator=administrator@somecompany.com
support=support@somecompany.co.uk

sales=sales@somecompany.com

- **<null> 요소**

<null>은 null 값을 다루기 위해 사용된다.

```
<bean class="SampleBean">
  <property name="email"><null/></property>
</bean>
```

위 코드는 Java Code의 sampleBean.setEmail(null)과 동일하다. 다음과 같이 정의한 경우에는 Java Code의 sampleBean.setEmail("")과 동일하다.

```
<bean class="SampleBean">
  <property name="email"><value></value></property>
</bean>
```

1.2.2.1.XML 기반의 설정 메타데이터 간략화

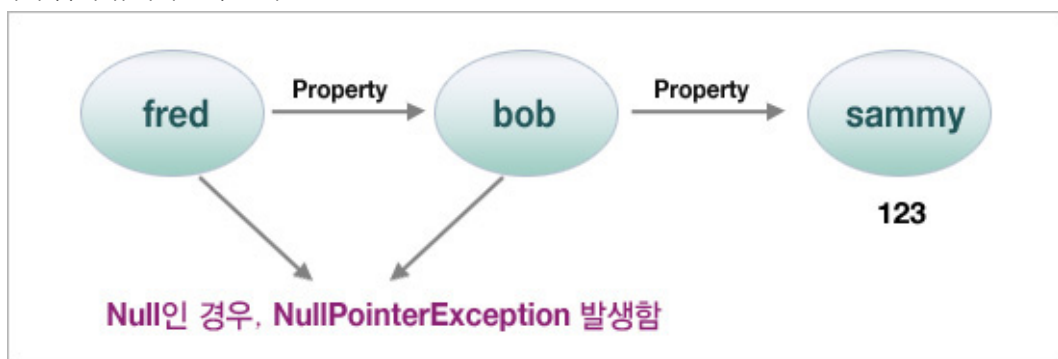
value나 Bean 참조를 위해 필요한 공통사항이다. 완전한 형태의 <value> 와 <ref>를 사용하는 것보다 간략화한 몇 가지 형태를 사용할 수 있다. <property>, <constructor-arg>, 그리고 <entry> 모두 완전한 형태의 <value> 요소 대신에 'value' 속성을 지원한다. 예를 들어 코드1이 코드2의 형태로 간략화 될 수 있다.

```
<!-- 코드 1 -->
<property name="myProperty"><value>hello</value></property>
```

```
<!-- 코드 2 -->
<property name="myProperty" value="hello"/>
```

1.2.2.2.혼합된 Property 명(Compound Property) - shortcut 기능 제공

복합적인 형태의 Property 정의가 가능하다. 마지막 Property명을 제외한 나머지 Property는 null이 아니어야 함에 유의하도록 한다.



```
<bean id="foo" class="foo.Bar">
  <property name="fred.bob.sammy" value="123" />
</bean>
```

위 예제에서 foo bean은 bob Property를 가지는 fred Property를 가진다. 그리고 bob Property는 sammy Property를 가지고 마지막 sammy Property는 123값으로 셋팅된다. 이렇게 되도록 하기 위해서는 foo의 fred Property, 그리고 fred의 bob Property는 bean이 생성된 후에 null이 아니어야만 한다. 그렇지 않으면 NullPointerException이 던져질 것이다.

1.2.3.depends-on 속성 사용

'depends-on' 속성은 Bean 이전에 초기화되어야 하는 하나 이상의 Bean을 명시적으로 강제하기 위해 사용된다. 다음은 depends-on 속성이 설정되어 있는 context-core.xml 파일의 일부이다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"
      autowire="byType" depends-on="coreMovieDao">
</bean>
```

다중 bean에 의존성을 표시할 필요가 있다면 아래의 예와 같이 콤마, 공백 그리고 세미콜론과 같은 모든 유효한 구분자를 사용하여 'depends-on'속성의 값으로 bean 이름 목록을 정의할 수 있다. 그러나 이 'depends-on' 속성을 사용하게 될 상황은 매우 드물다.

```
<bean id="beanOne" class="SampleBean" depends-on="manager,accountDao">
  <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

위의 예제는 beanOne Bean이 생성되기 이전에 manager Bean이 생성되어 특정 서버를 구동시켜놓거나 특정 리소스에 대한 작업을 수행해놓고 있어야 beanOne Bean이 정상적으로 동작하므로 강제적으로 manager Bean을 초기화시킨다.

1.2.4.Lazy Instantiation

기본적으로 Spring IoC Container가 Start될 때 singleton Bean에 대해서는 모두 인스턴스화한다.

- 특정 singleton Bean을 Container가 Start될 때 인스턴스화 시키지 않고 처음 Bean 요청이 들어왔을 때 인스턴스화 시키고자 하면 'lazy-init' 속성을 설정한다. 다음은 Lazy Instantiation 속성이 설정되어 있는 파일인 context-core.xml 파일의 일부이다.

```
<bean id="coreMovieDao"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDaoImpl" lazy-init="true"/>
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"/>
```

- 모든 Bean들에 대해서 기본적으로 Lazy 인스턴스화 시키고자 하면 'default-lazy-init' 속성을 설정하면 된다.

```
<beans default-lazy-init="true">
  <!-- no beans will be eagerly pre-instantiated -->
</beans>
```

1.2.5.Autowiring

Spring IoC Container는 Bean들 사이의 관계를 autowire 할 수 있다. 이것은 BeanFactory의 내용을 조사함으로써 Spring이 자동적으로 협력자(다른 bean)를 분석하는 것이 가능하다는 것을 의미한다. autowiring을 사용하면 명백하게 많은 양의 타이핑을 줄이고 Property나 생성자의 인자를 명시할 필요를 줄이거나 제거하는 것이 가능해진다. XML-기반의 설정 메타데이터를 사용할 때, Bean정의의 위 autowire 모드는 <bean>의 autowire 속성을 사용하면 된다. <bean>의 autowire 속성에 정의할 수 있는 값은 다음과 같다.

속성	설명
no	[기본 설정] Autowiring 기능 사용 안 함
byName	Property명과 동일한 id나 name을 가진 Bean을 찾아 Autowiring 기능 적용
byType	해당 Property 타입의 Bean이 하나 존재한다면 Autowiring되나 하나 이상 존재 시 UnsatisfiedDependencyException 발생됨. 만약 대응되는 Bean이 없다면 Property 셋팅 안됨
constructor	이것은 byType과 유사하지만 생성자의 인자에 적용됨. BeanFactory 내 생성자의 인자 타입과 맞는 Bean이 정확하게 하나가 아닐 경우 UnsatisfiedDependencyException 발생됨
autodetect	constructor 모드 수행 후 byType 모드가 수행됨
default	<beans>의 default-autowire 속성에 설정한 autowire 모드가 해당 Bean에 적용됨

다음은 Autowiring 속성이 설정되어 있는 context-core.xml 파일의 일부이다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"
      autowire="byType" depends-on="coreMovieDao">
</bean>
```

1.2.5.1. 장점

- Property나 생성자의 인자를 XML에 설정할 필요 없음
- XML 파일 크기 줄어듦
- 참조 관계에 있는 타 Bean들의 변경 및 추가 시 XML 파일의 변경이 최소화됨
- 동일한 이름의 Bean을 XML에 중복 정의하여 사용하는 혼동을 없애 줌

1.2.5.2. 단점

- Bean들의 관계가 명시적으로 문서화되지 않음으로써 기대되지 않는 결과를 가지지 않게 주의해야 함
 - 타입에 의한 Autowiring은 잠재적인 모호함을 가져올 수 있음
- * Autowiring 대상에서 특정 Bean을 제외하려면 autowire-candidate 속성을 false로 설정해주어야 한다.

```
<bean id="bean" class="sample.TestBean" autowire-candidate="false" />
```

1.2.6. Dependency Check

해당 Bean에 설정된 모든 Property들(Primitive Type/Collection 및 Bean 참조)이 제대로 설정되었는지 확인한다.

- <bean>의 dependency-check 속성 설정

모드	설명
none	[기본 설정] 의존성 확인 안 함. 참조관계의 Bean이 존재하지 않는 경우 Property 설정 안 함
simple	Primitive Type과 collection을 위해 의존성 확인 수행
object	참조관계의 Bean을 위해 의존성 확인 수행
all	simple과 object 모드를 모두 수행

다음은 Dependency Check의 속성 정의 예시이다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin....MovieServiceImpl" dependency-check="object">
    <property name="coreMovieDao" ref="coreMovieDao" />
</bean>
```

또한 다음과 같은 방법으로 모든 Bean들에 대해서 동일하게 Dependency Check 여부를 설정할 수 있다.

```
<beans default-dependency-check="none">
    <!-- no beans will be eagerly pre-instantiated -->
</beans>
```

1.3.Method Injection

Dependency Injection의 방법인 setter injection과 constructor injection을 사용할 경우, Singleton Bean은 참조하는 Bean들을 Singleton 형태로 유지하게 된다. 그런데 특별한 경우에는 Singleton Bean이 Non Singleton Bean(즉, Prototype Bean)과 Dependency 관계를 가질 수 있다. 이 같은 상황이 발생할 때 Lookup Method Injection을 사용하여 해결하는 것이 가능 하다. 동일한 상황에서 BeanFactoryAware를 구현하여 해결하는 방법도 존재하나 Spring Container API에 종속적으로 Bean 코드가 변경되므로 바람직한 해결 방법이 아니다.

- Lookup Method Injection
- Method Replacement

1.3.1.Lookup Method Injection

Singleton Bean이 Prototype Bean을 참조해야 할 경우 <lookup-method>를 설정한다. 다음은 Lookup Method Injection을 이용하여 참조 관계를 정의한 context-core.xml 의 일부이다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    <!-- method injection -->
    <lookup-method name="getMovieDao" bean="coreMovieDao"/>
</bean>

<!-- change scope from singleton to prototype (non singleton) -->
<bean id="coreMovieDao"

      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao" scope="prototype"/>
```

해당 lookup 메소드는 다음과 같이 MovieDao 리턴하는 형태로 메소드를 구현하도록 한다.

```
public class MovieServiceImpl ... {
    public MovieDao getMovieDao(){
        // do nothing - this method will be overridden by Spring Container
        return null;
    }
    중략...
}
```

1.3.2.Method Replacement

이미 존재하는 기존의 메소드를 수정하지 않은 상태에서 메소드의 기능을 변경하고자 할 때 <replaced-method>를 이용한다. 사용 예제는 다음과 같다.

- 구현 클래스

Spring Framework에서 제공하는 MethodReplacer 인터페이스를 구현한 클래스를 생성하고, reimplement 메소드 내에 로직을 구성한다.

```
import org.springframework.beans.factory.support.MethodReplacer;
public class SayHelloMethodReplacer implements MethodReplacer{
    public Object reimplement (Object target, Method method, Object[] args)
        throws Throwable {
        종략...
```

- 속성 정의 파일

```
<bean id="beanFirst" class="test.BeanFirst"/>
<bean id="beanSecond" class="test.BeanSecond">
    <replaced-method name="sayHello" replacer="methodReplacer">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>
<bean id="methodReplacer" class="test.SayHelloMethodReplacer"/>
```

위 속성 정의 파일에서는 BeanSecond 클래스의 sayHello 메소드 실행 시점에, 앞서 정의한 MethodReplacer가 적용되도록 정의하고 있음을 알 수 있다.

1.4.Bean과 Container의 확장

Spring Framework의 Container는 기본적으로 확장이 되도록 설계되어 있다. 모든 어플리케이션 개발자들이 확장하여 사용할 필요는 없고 확장할 필요성이 있는 경우에 확장하여 사용하도록 한다. 다음 각각의 항목 별로 기본적으로 제공되는 내용과 확장하여 사용할 수 있는 내용을 설명한다.

- Bean Scope
- Bean Life Cycle
- Bean 상속
- Container 확장
- ApplicationContext 활용

1.4.1.Bean Scope

Spring Framework에서 지원하는 5가지 Scope에 따라 Bean의 인스턴스 생성 메커니즘이 결정된다. 서비스 Scope은 설계, 개발 단계에서 결정하기 어려우므로, 기본적으로는 Default Scope인 Singleton으로 개발하고, 추후 해당 서비스의 성격에 따라 Scope을 정의하는 것이 좋다.

- <bean>의 scope 속성값

속성	설명
singleton	[기본 설정] Spring IoC Container 내에서 Bean 정의 당 하나의 Bean 객체 생성
prototype	매번 같은 Type의 새로운 Bean 객체 생성
request	WebApplicationContext 유형의 Container 사용 시, Http request 당 하나의 Bean 객체 생성
session	WebApplicationContext 유형의 Container 사용 시, Http session 당 하나의 Bean 객체 생성

속성	설명
globalSession	WebApplicationContext 유형의 Container 사용 시, portlet context 내에서만 유효하며 global Http session 당 하나의 Bean 객체 생성

이 외에도, custom scope을 통해 신규 Scope에 대해 정의할 수 있다.

1.4.1.1.Singleton

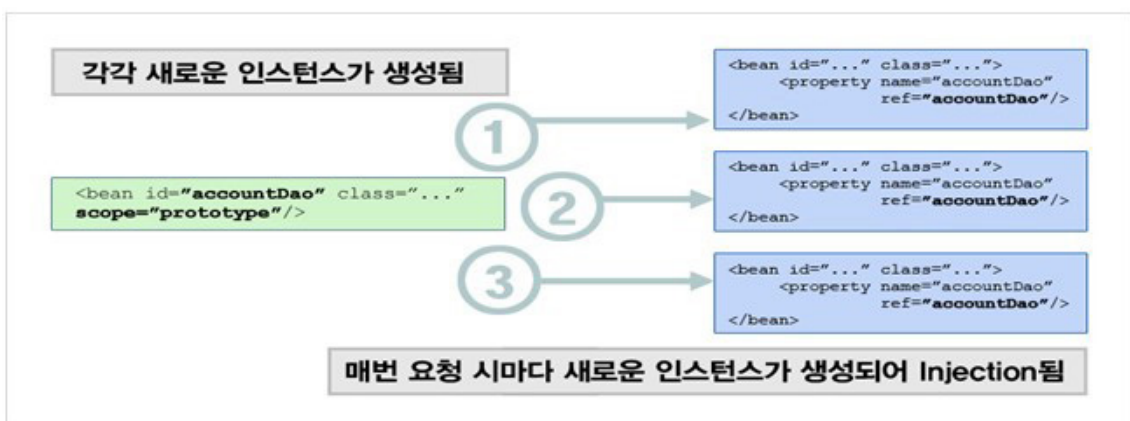


Singleton Scope은 기본 Scope으로 여러 개의 요청에 대해 하나의 Bean 인스턴스를 생성하여 제공한다. 따라서 Client Request마다 유지해야 하는 Data가 있다면, Singleton Scope의 서비스는 적합하지 않다. 다음은 Singleton Scope의 속성 정의 예시이다.

```
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"
      scope="singleton">
  <property name="coreMovieDao" ref="coreMovieDao" />
</bean>
<bean id="coreMovieDao"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao">
  종 략...
</bean>
```

위와 같이 singleton scope을 정의 할 수 있지만 scope의 기본 설정값이 singleton이므로 따로 정의해야 할 필요가 없다.

1.4.1.2.Prototype



Prototype Scope은 요청시마다 Bean 인스턴스를 생성하여 제공한다. 따라서 여러 Client가 동시에 한 Bean 인스턴스에 접근할 수 없다. 다음은 Prototype Scope의 속성 정의 예시이다.

```
<bean id="coreMovieService"
```

```

class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"
scope="prototype">
  <property name="coreMovieDao" ref="coreMovieDao" />
</bean>

```

※ 일반적으로 인스턴스의 Singleton 여부를 판단하기 위해서 전역변수의 존재 여부를 이용한다. 즉, 전역변수가 존재하지 않은 인스턴스의 경우에는 Singleton, 전역변수가 존재하는 경우에는 Prototype 으로 정의할 수 있다. 그러나 해당 전역변수가 read-only인지 writable 가능한지에 따라서 이 같은 구분은 변경될 수 있다. 따라서 인스턴스를 Singleton으로 생성할지 Prototype으로 생성할지에 대한 여부에 대해서는 개발자들이 해당 Scope의 인스턴스가 메모리에서 어떻게 사용되는지를 이해하는 것이 가장 좋다.

- Singleton
 - Shared objects with no state
 - Shared object with read-only state
 - Shared object with shared state : 이 경우에는 Synchronization을 적절하게 사용하여 동시성을 제어하도록 해야 한다.
 - High throughput objects with writable state : 일반적으로 Object Pooling과 같은 기능을 사용하는 것을 예로 들 수 있다. 인스턴스를 생성하는데 많은 비용이 발생하거나 무수히 많은 인스턴스를 관리할 필요가 있는 경우에는 Object Pooling을 사용하고 Pooling 대상이 되는 인스턴스는 Singleton으로 사용할 수 있다. 이 경우에도 Writable State에 변경이 발생할 때 Synchronization을 적절하게 사용해야 한다.
- Prototype
 - Objects with writable state
 - Objects with private state

1.4.1.3.Other Scopes

request, session, globalSession Scope 사용 시 주의 사항은 다음과 같다.

- Web 기반의 ApplicationContext 사용시에만 이 Scope들을 사용할 수 있으며 그 외의 경우 사용하게 되면 IllegalStateException이 발생한다.
- Scope이 다른 Bean에서 참조하는 경우 Bean 정의 시<aop:scoped-proxy/>와 함께 작성해야 한다.(아래의 예시 참고)

moviePreferences Bean은 scope이 session이지만 coreMovieService Bean의 scope이 singleton(default가 singleton)이기 때문에 문제가 발생한다. 즉, 매 세션마다 moviePreferences 객체를 만들어줘야 하지만 coreMovieService Bean에 의해 MoviePreferences 객체가 한 번만 생성되기 때문에 원하던 대로 동작하지 못하는 것이다. 따라서 매 세션 마다 새로운 객체를 만들어서 줄 Proxy를 만들기 위해서 <aop:scoped-proxy/>를 사용하도록 한다.

```

<!-- a HTTP Session-scoped bean exposed as a proxy -->
<bean id="moviePreferences"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MoviePreferences"
      scope="session">
  <!-- this next element effects the proxying of the surrounding bean -->
  <aop:scoped-proxy/>
</bean>
<!-- a singleton-scoped bean injected with a proxy to the above bean -->
<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
  <!-- a reference to the proxied 'moviePreferences' bean -->
  <property name="moviePreferences" ref="moviePreferences"/>
</bean>

```

1.4.1.4.Custom

신규 Scope을 정의하기 위한 클래스를 생성하고, org.springframework.beans.factory.config.Scope 인터페이스를 implements한다. 또한 CustomScopeConfigurer를 이용하여 신규 정의한 Custom Scope을 등록하여 Custom Scope를 사용할 수 있도록 한다.

해당 프로젝트에 적합한 Scope을 아래의 예시와 같이 직접 정의할 수 있다.

```
<!-- 신규 Scope 정의할 위한 클래스를 정의하고,  
org.springframework.beans.factory.config.Scope 인터페이스를 implement한다.-->  
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">  
  <!-- CustomScopeConfigurer를 이용하여 Custom Scope 등록 -->  
  <property name="scopes">  
    <map>  
      <entry key="thread">  
        <bean class="com.foo.ThreadScope"/>  
      </entry>  
    </map>  
  </property>  
</bean>  
  
<!-- Custom Scope 사용 -->  
<bean id="bar" class="x.y.Bar" scope="thread">  
  <property name="name" value="Rick"/>  
  <aop:scoped-proxy/>  
</bean>
```

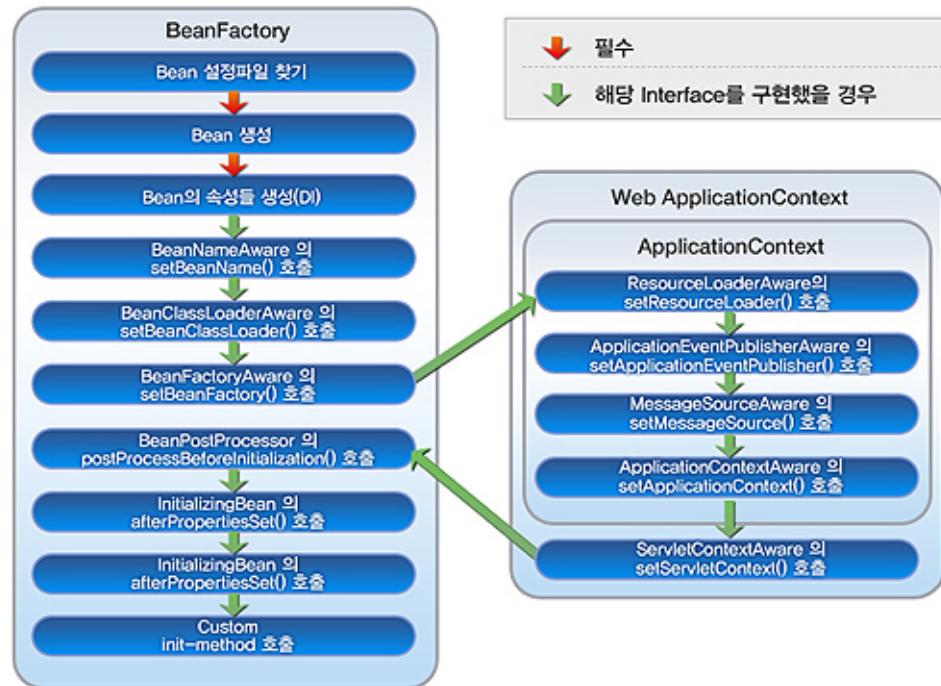
1.4.2.Bean Life Cycle

Bean의 Life Cycle은 다음 그림에서와 같이 Initialization, Activation, Destruction으로 구성된다.



1.4.2.1.Initialization

Spring Container는 아래 그림에서 보여지는 여러 과정을 통해 구동된다. Spring Bean 클래스가 아래 그림에서 보여지는 각각의 인터페이스들을 구현하였을 때 각각의 메소드들이 호출된다.



Spring Framework에서 지원하는 Life Cycle 메소드를 그대로 사용할 경우 특정한 인터페이스를 구현해야 하므로, 해당 코드가 Spring Framework에 의존적일 수 있게 된다. 즉, 위 그림에서 제시하고 있는 Life Cycle 메소드를 사용하기 위해서는 Spring Bean 클래스에서 해당 Life Cycle 인터페이스 클래스를 구현해줘야 한다. 예를 들어, ApplicationContextAware 인터페이스 클래스를 구현한 Spring Bean에서는 setApplicationContext(ApplicationContext context) 메소드를 작성하고, Spring Bean 내부에서 ApplicationContext를 이용하여 ApplicationContext에서 제공하는 메소드를 호출할 수 있다.

```
public class IoServiceImpl implements IoService1,
    ApplicationContextAware {
    public void setApplicationContext (ApplicationContext context){
        IoService2 iocService2 = (IoService2)context.getBean("IoService2");
    }
}
```

또다른 예로 MessageSourceAware 인터페이스 클래스의 경우, Spring Container에 정의된 MessageSource를 얻기 위해 사용될 수 있다. MessageSourceAware 인터페이스 클래스를 구현한 Spring Bean에서 setMessages(MessageSource messages) 메소드를 작성하여 MessageSource에 접근할 수 있다.

```
public class IoServiceImpl implements IoService1, MessageSourceAware {
    private MessageSource messageSource;
    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
}
```

이와는 달리 Bean 속성(init-method, destroy-method) 정의를 통해 특정 인터페이스에 대한 구현없이 별도 Life Cycle 메소드를 정의할 수도 있다. 다음은 init-method 속성이 정의된 context-core.xml의 일부이다.

```
<bean id="coreMovieService"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"
    init-method="movieInitialize" destroy-method="movieDestroy" parent="parent">
</bean>
```

모든 Bean에 대한 초기화 method 설정은 <beans>의 default-init-method 속성을 이용하도록 한다.

1.4.2.2.Destruction

Destruction 단계에서는 BeanFactory와 ApplicationContext가 동일하게 동작한다.



다음은 destroy-method 속성이 정의된 context-core.xml의 일부이다.

```

<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"
      init-method="movieInitialize" destroy-method="movieDestroy"
      parent="parent">
</bean>
  
```

모든 Bean의 소멸자 method 설정은 <beans>의 default-destroy-method 속성을 이용한다.

1.4.3.Bean 상속

Bean 정의는 여러 속성 정보들, 생성자 인자, Property 값을 포함하여 많은 양의 설정 정보를 포함한다. 자식 Bean은 부모 정의로부터 설정 정보를 상속하여 정의한다. 그러므로 값을 오버라이드하거나 다른 것을 추가할 수 있다. 상속 관계를 이용하여 Bean을 정의하는 것은 XML 파일의 양을 줄일 수 있으므로 템플릿 형태의 부모 Bean을 정의하는 것은 유용하다. XML 기반의 속성 정의시 자식 Bean은 부모 Bean을 명시하기 위해 'parent' 속성을 사용해야 한다.

- 부모 Bean 정의

특수 설정 없이 부모 Bean으로 사용이 가능하며 class 속성 값을 설정하지 않은 경우, 반드시 abstract 속성 값을 "true"로 설정한다. abstract 속성 값이 "true"인 경우 Bean의 인스턴스화가 불가능하다.

- 자식 Bean 정의

parent 속성 값에 부모 Bean의 id 혹은 name을 설정한다.

다음은 Bean 상속이 표현되어 있는 context-core.xml의 일부이다.

```

<!-- register parent bean that has a dependency with coreMovieDao bean -->
<bean id="parent" abstract="true">
  <property name="coreMovieDao" ref="coreMovieDao" />
</bean>

<bean id="coreMovieService"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl"
      init-method="movieInitialize" destroy-method="movieDestroy"
      parent="parent">
</bean>
  
```

1.4.4.Container 확장

1.4.4.1.Bean 후처리

Bean의 LifeCycle 중 Initialization 단계에서 Bean 초기화 시점 전후에 수행되는 것을 **Bean 후처리**라고 하며, BeanPostProcessor를 구현하면 기능을 확장할 수 있다. ApplicationContext 유형의 Container 사용 시에는 XML 파일에 BeanPostProcessor 인터페이스를 구현한 클래스를 등록만 시키면 Container가 해당 클래스를 BeanPostProcessor로 인식하여 각각의 Bean을 초기화하기 전과 후에 후처리 메소드를 호출해 준다. 그러나 BeanFactory 유형의 Container를 사용하고 있다면 BeanFactory의 addBeanPostProcessor() 메소드를 이용하여 프로그램 상에서 등록해야 한다. 예시는 다음과 같다.

```
public class InstantiationTracingBeanPostProcessor
    implements BeanPostProcessor {
    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        return bean; // we could potentially return any object reference here
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }
}
```

```
<bean class="scripting.InstantiationTracingBeanPostProcessor"/>
```

1.4.4.2.BeanFactory 후처리

BeanFactoryPostProcessor를 구현하여 BeanFactory 후처리 기능을 확장할 수 있다. 모든 Bean에 대한 정의가 로딩된 후, BeanPostProcessor Bean을 포함한 어떤 Bean이라도 인스턴스화되기 이전에 Spring Container에 의해 BeanFactoryPostProcessor의 postProcessBeanFactory() 메소드가 호출된다. 따라서, BeanFactoryPostProcessor 인터페이스를 구현한 클래스 내에서 postProcessBeanFactory 메소드를 작성하고, Bean으로 정의하면 된다. 예시는 다음과 같다.

```
public class BeanCounterBeanFactoryPostProcessor
    implements BeanFactoryPostProcessor {
    public void postProcessBeanFactory(ConfigurableListableBeanFactory factory)
        throws BeansException {
        중략...
    }
}
```

```
<bean class="test.BeanCounterBeanFactoryPostProcessor"/>
```

BeanFactoryPostProcessor는 BeanFactory 유형의 Container와 함께 사용될 수 없다. 유용한 BeanFactoryPostProcessor 구현 클래스는 PropertyPlaceholderConfigurer와 CustomEditorConfigurer이다.

다음은 PropertyPlaceholderConfigurer와 CustomEditorConfigurer에 대한 사용 예이다.

- 설정 정보의 외부화

PropertyPlaceholderConfigurer를 사용하여 하나 이상의 외부 Property 파일로부터 속성들을 로딩하고 그 속성들을 이용하여 Bean 정의 XML 파일에서의 위치소유자(placeholder) 변수들을 채운다.

다음은 설정 정보 외부화를 위해 PropertyPlaceholderConfigurer 클래스를 Bean으로 등록하고 있는 context-core.xml 의 속성 정의 부분이다.

```
<!-- set file locations -->
<bean id="configurer"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>MovieConfigurer.properties</value>
        </list>
    </property>
</bean>

<bean id="coreMovieService"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    <property name="coreMovieDao" ref="coreMovieDao" />
    <!-- set movieTitle value using key name in properties file -->
    <property name="movieTitle" value="${movie.title}"></property>
</bean>
```

위에서 외부 파일로 정의된 movieConfigurer.properties 의 내용은 다음과 같다.

```
movie.title=Shrek
```

• PropertyEditor 확장

CustomEditorConfigurer를 사용하여 java.beans.PropertyEditor의 커스텀 구현 클래스를 등록하여 특성 값을 다른 특성 타입으로 번역할 수 있도록 한다. 확장한 PropertyEditor 클래스를 속성 정의 파일에 등록 후 PropertyEditor로 사용한다.

```
<bean id="customEditorConfigurer"
    class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="com.springinaction.knight.PhoneNumber">
                <bean id="phoneEditor"
                    class="com.springinaction.springcleaning.PhoneNumberEditor" />
            </entry>
        </map>
    </property>
</bean>
```

```
<bean id="knight" class="com.springinaction.knight.KnightOnCall">
    <property name="url" value="http://www.knightoncall.com" />
    <property name="phoneNumber" value="940-555-1234" />
</bean>
```

1.4.5.ApplicationContext 활용

1.4.5.1.MessageSource를 활용한 국제화(I18N) 지원

ApplicationContext 인터페이스는 MessageSource라고 불리는 인터페이스를 확장해서 메시징(국제화 지원)기능을 제공하며 HierarchicalMessageSource와 함께 구조적인 메시지를 분석하는 능력을 가진다. MessageSourceAware인터페이스를 구현하는 Bean은 ApplicationContext의 messageSource Bean을 사용할 수 있다.

다음은 context-common.xml 의 messageSource 속성 정의 부분이다.

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list><value>message/message-moviemgmt</value></list>
  </property>
</bean>
```

Resource Bundle 파일은 국제화 지원을 위해 Locale별 파일로 구성하며 위에서 참조하는 properties 파일은 다음과 같다.

```
errors.required={0} is a required field.
```

또한 messageSource를 얻는 부분은 다음과 같이 구현되어 있다.

```
new String(messageSource.getMessage("errors.required", new Object[] { "TITLE" },
    Locale.KOREA).getBytes("8859_1"), "euc-kr")
```

messageSource 부분을 테스트 할수있는 파일을 수행시키면 다음과 같은 message를 확인할 수 있다.

```
"TITLE" 필드는 반드시 필요 하다.
```

1.4.5.2.Event

ApplicationContext는 어플리케이션이 구동하는 동안 다수의 이벤트를 발생시킬 수 있으므로, Listener를 Bean으로 등록하게 되면, Container는 해당하는 Event가 발생하면 관련 Listener의 onApplicationEvent() 메소드를 호출한다.

• Built-in Events

이벤트	설명
ContextRefreshedEvent	ApplicationContext가 초기화되거나 갱신(refresh)될 때 발생하는 이벤트 - 여기서 초기화는 모든 Bean이 로드되고 Singleton Bean들은 미리 인스턴스화되며 ApplicationContext는 사용할 준비가 된다는 것을 의미함
ContextClosedEvent	ApplicationContext의 close()메소드를 사용하여 ApplicationContext가 종료될 때 발생하는 이벤트 - 여기서 종료는 Singleton Bean들이 소멸(destroy)되는 것을 의미함
RequestHandledEvent	HTTP Request가 처리되었을 때 WebApplicationContext 내에서 발생하는 이벤트 - 이 이벤트는 Spring의 DispatcherServlet을 사용하는 웹 어플리케이션에서만 적용 가능함

ApplicationListener를 구현한 Listener의 예시는 다음과 같다.

```
public class RefreshListener implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent evt) {
        if (evt instanceof ContextRefreshedEvent) {
            종 략...
        }
    }
}
```

앞서 구현한 RefreshListener 클래스에 대한 속성 정의 예시는 다음과 같다.

```
<bean id="refreshListener" class="sample.RefreshListener"/>
```

• Custom Event 발생

사용자 정의 Event를 직접 발생시키고 해당 Event 발생 시 처리될 수 있도록 Listener를 등록하는 것도 가능하다. Event Listening을 하기 위해서는 Listener 등록이 필요하다. 다음은 Listener Bean을 등록하는 context-core.xml 파일의 일부이다.

```
<bean id="movieEventListener"
      class="org.anyframe.plugin.core.moviefinder.service.impl.MovieEventListener"/>
```

다음은 Custom Event인 MovieEvent를 처리하고 있음을 알 수 있다.

```
public class movieEventListener implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent evt) {
        if (evt instanceof MovieEvent) {
            MovieEvent event = (MovieEvent)evt;
            System.out.println("Received in MovieEventListener : " +
                               event.getMovieMessage());
        }
    }
}
```

다음은 Custom Event인 MovieEvent를 발생시키는 부분이다.

```
this.ctx.publishEvent(new MovieEvent(this,"new movie is added successfully."));
```

1.4.5.3.BeanFactory와 ApplicationContext 특징 비교

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Automatic BeanPostProcessor registration	No	Yes
Automatic BeanFactoryPostProcessor registration	No	Yes
Convenient MessageSource access (for i18n)	No	Yes
ApplicationEvent publication	No	Yes

대부분의 전형적인 어플리케이션 구축 시에는 ApplicationContext 사용을 권장한다.

1.5.XML 스키마 기반 설정

XML 스키마에 기초하여 새로운 XML 설정 문법이 나오고 있으며 점점 더 쉽게 XML을 설정할 수 있도록 Spring Framework은 진화하고 있다. 또한 XML 스키마를 확장하여 사용할 수도 있다.

- 기본으로 제공되는 XML 스키마

Spring Framework에서 기본으로 제공하는 XML 스키마의 종류는 다음과 같다.

[util, jee, lang, jms, tx, aop, context, tool, beans] (각각의 사용법은 Spring 매뉴얼 사이트 [<http://static.springsource.org/spring/docs/4.0.x/spring-framework-reference/html/xsd-config.html>] 를 참고하도록 한다.)

- XML 스키마 확장 가능

어플리케이션 개발 시 어플리케이션 도메인을 좀더 잘 표현할 자체적인 도메인 속성의 설정 태그를 정의할 수 있다.

확장한 스키마를 실제 XML 파일에 적용하여 사용하는 방법은 Spring 매뉴얼 사이트 [<http://static.springsource.org/spring/docs/4.0.x/spring-framework-reference/html/extensible-xml.html>]를 참고하도록 한다.

- XML 스키마 참조 방법

xmlns:~를 이용하여 사용하고자 하는 namespace를 정의하고, 해당 namespace의 XML 스키마를 정의한 XSD 파일의 location을 정의한다.

```
<?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:util="http://www.springframework.org/schema/util"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:lang="http://www.springframework.org/schema/lang"
  xmlns:jms="http://www.springframework.org/schema/jms"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-4.0.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-4.0.xsd
    http://www.springframework.org/schema/lang
    http://www.springframework.org/schema/lang/spring-lang-4.0.xsd
    http://www.springframework.org/schema/jms
    http://www.springframework.org/schema/jms/spring-jms-4.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">
  <!-- <bean/> definitions here -->
</beans>
```

- XML 설정에 대한 부담시 Annotation 활용 제안

XML 기반에서 Bean을 정의하는 방식 외에 Annotation을 활용하면 XML 설정에 대한 부담을 덜 수 있다.

2.Annotation

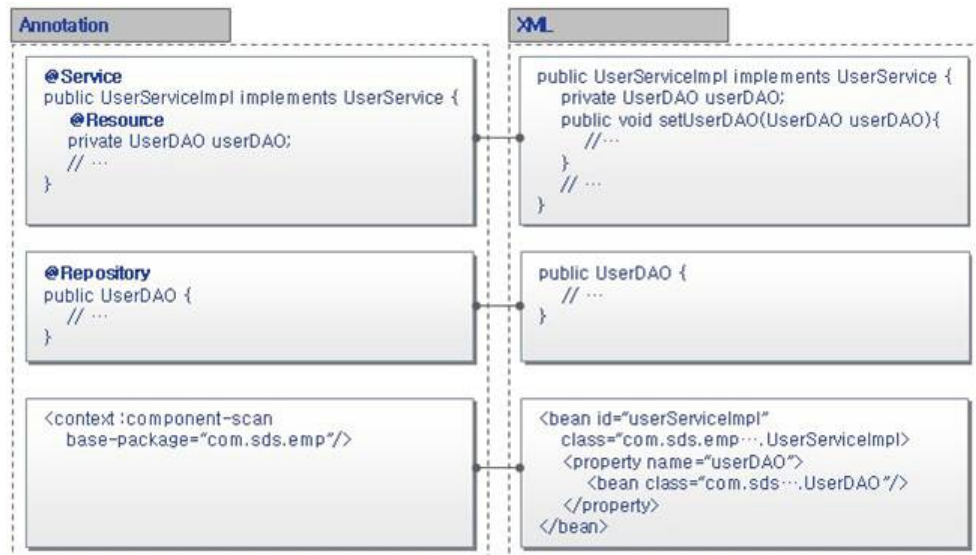
Spring XML 만을 독립적으로 사용할 경우 때때로 방대하고 복잡한 속성 파일들로 인해 시스템 개발 및 유지보수의 지연을 초래할 가능성이 높아진다. 이러한 문제점을 해결하기 위해 Spring Framework 에서는 별도 XML 정의없이도 사용 가능한 annotation 지원에 주력하고 있는 실정이다. Spring 2.0 에서는 @Transactional, @Required, @PersistenceContext / @PersistenceUnit과 같은 Transaction 관리 또는 Persistence 관리 영역에 대한 annotation들을 지원했다면 Spring 2.5부터는 Bean 또는 Dependency 정의 등과 같이 Spring 속성 정의 XML과 직접적으로 관련된 annotation들을 선보이고 있다. 또한 Spring 3에서는 Spring 특화된 Annotation 외에 Dependency Injection에 관한 표준 Annotation 인 JSR-330(Dependency Injection for Java) Annotation 사용을 지원하기 시작했다. 본 문서에서는 annotation 사용 용도를 Bean Management, Dependency Injection, Life Cycle로 구분하고 각각의 경우에 따른 사용법에 대해 상세히 살펴해보도록 하자.

기본적으로, Annotation은 JDK 1.5 이상에서 활용이 가능하며, Spring Container가 Annotation을 인식할 수 있도록 하기 위해서는 속성 정의 XML 파일 내에 다음과 같은 정의가 추가되어야 함에 유의해야 한다.

```
<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-4.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">
  <context:annotation-config/>
</beans>
```

- **XML vs. Annotation**

다음은 특정 서비스를 구성하는 구현 클래스, DAO 클래스, 속성 정의 XML에 대해 XML을 이용하는 경우와 Annotation을 이용하는 경우로 나누어 비교해 본 그림이다.



2.1.Bean Management

Stereotype Annotation을 사용하면 Spring Framework의 컨테이너에 의해 관리되어야 하는 Bean들을 정의할 수 있다. 일반적으로 Parent Stereotype Annotation인 @Component를 활용하면 모든 Bean에 대한 정의가 가능하다. 그러나 Spring Framework에서는 레이어별로 구성 요소를 구분하여 다음과 같은

Annotation을 사용할 것을 권장하고 있고, 향후 지속적으로 레이어별 특성을 반영할 수 있는 속성들을 추가해 나아갈 예정이다.

- **@Service**

비즈니스 로직을 처리하는 클래스를 정의하는데 사용한다.

- **@Controller**

프리젠테이션 레이어를 구성하는 Controller 클래스를 정의하는데 사용하며, Spring MVC 기반인 경우에 한해 활용 가능하다.

- **@Repository**

데이터 접근 로직을 처리하는 클래스를 정의하는데 사용하며, 퍼시스턴스 레이어에서 발생한 Exception에 대한 Translation이 지원된다.



[참고] Persistence Layer Exception Translation

DAO 구현 시에 Hibernate, JPA, 또는 JDO 같은 프레임워크들을 사용할 경우 각 기술들이 사용하는 고유의 Exception(예: HibernateException, PersistenceException, JDOException 등)이 Run-time시에 발생할 수 있다. Spring에서는 'PersistenceExceptionTranslator'를 제공하여 Data Access 프레임워크 고유 Exception을 Spring의 DataAccessException 타입의 Exception으로 변환해줌으로써, 어플리케이션 코드에서 특정 Data Access 프레임워크의 Exception API에 종속되지 않고 일관성 있게 Exception 처리를 할 수 있도록 도와준다.

JSR-330에서는 컴포넌트 식별을 위해 @Named Annotation을 제공하고 있으며 Spring 3에서는 특정 Bean 클래스에 대해 @Named를 부여한 경우 Stereotype Annotation을 부여한 경우와 마찬가지로 컨테이너에 의해 해당 Bean이 관리될 수 있도록 지원한다. 단, @Named를 부여한 Bean에 대해서는 기본 Scope인 'Singleton'으로 적용되며 다른 유형의 Scope 처리는 향후 릴리즈 시에 반영될 예정이다.

본 문서에서는 위에서 나열한 annotation을 사용하는 방법에 대해서 자세히 살펴보도록 한다.

2.1.1.Auto Detecting

Stereotype Annotation을 사용하여 Bean을 정의하면 XML에 따로 Bean 정의를 명시하지 않아도 Spring Container가 Bean을 인식하고 관리할 수 있다. 단, 자동 인식이 되기 위해서는 서비스 속성 정의 XML 내에 **<context:component-scan/>** 을 정의해 주어야 한다. 이 설정을 추가하면 Spring Container는 클래스패스 상에 존재하는 클래스들을 스캔하여 Stereotype Annotation이 정의된 클래스들 Bean으로 인식하고 자동으로 등록한다.

```
<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-4.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">
  <context:component-scan base-package="org.anyframe.plugin" />
</beans>
```

<context:component-scan />을 정의한 경우 Annotation 인식을 위한 설정 **<context:annotation-config/>** 을 별도로 추가하지 않아도 된다.

다음은 서비스 레이어의 구성 요소인 MovieServiceImpl 클래스에 대해 @Service라는 Stereotype Annotation을 사용한 예이다.

```
@Service
public class MovieServiceImpl implements MovieService {
    @Inject
    @Named("coreMovieDao")
    private MovieDao movieDao;
}
```

위 예제에서는 해당 클래스의 클래스명(소문자로 시작)이 Bean name으로 셋팅되어 해당 Bean을 찾을 때 **movieServiceImpl**이라는 문자열을 사용해야 한다.

```
MovieService service = (MovieServiceImpl) context.getBean("movieServiceImpl");
```

해당 Annotation에 속성을 부여하면, 원하는 Bean name을 지정하는 것 또한 가능하다.

```
@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Inject
    @Named("coreMovieDao")
    private MovieDao movieDao;
}
```

이 경우에 해당되는 Bean을 찾기 위해서는 속성으로 정의한 Name을 활용해야 한다.

```
MovieService service = (MovieServiceImpl) context.getBean("movieServiceImpl")
```

2.1.2.Using Filters to customize scanning

<context:component-scan>의 여러 속성들을 이용하면 검색 대상의 범위를 조정하여 자동으로 검색되어 Bean으로 등록되는 클래스들을 filtering 할 수 있다. base-package 속성은 <context:component-scan> 내에 정의 가능한 속성으로 검색 대상 패키지를 정의하는 용도로 사용된다. 이외에도 <context:component-scan>은 하위 element로 <context:include-filter>, <context:exclude-filter>를 가질 수 있는데, 다양한 Filter Type(type)에 해당하는 표현식(expression)을 정의함으로써 이에 해당하는 클래스들을 포함 또는 제외시킬 수가 있다. 다음은 <context:include-filter>, <context:exclude-filter> 사용 예이다.

```
<context:component-scan base-package="org.anyframe.plugin">
    <context:include-filter type="regex" expression=".*Stub.*Repository">
    <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

정의 가능한 Filter Type은 4가지이며, 다음과 같다.

Filter Type	Example Expressions
annotation	org.example.SomeAnnotation
assignable	org.example.SomeClass
regex	org\example\.Default.*
aspectj	org.example..*Service+



참고

Bean 정의를 위해 Annotation을 부여한 클래스를 auto detection하는 디폴트 설정을 사용하지 않고자 하는 경우에는 <context:component-scan />태그에 **use-default-filters="false"** 속성을 추가하면 된다.

2.1.3.Scope Definition

Spring Framework에서는 Bean의 인스턴스 생성 메커니즘에 따라 5가지 Scope 을 제공하는데 이러한 Bean Scope을 정의하기 위해서는 다음과 같이 @Scope을 사용하도록 한다.

```
@Scope("prototype")
@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Inject
    @Named("coreMovieDao")
    private MovieDao movieDao;
}
```

2.2.Dependency Injection

특정 Bean의 기능 수행을 위해 다른 Bean을 참조해야 하는 경우 사용하는 Annotation으로는 @Autowired, @Resource 그리고 @Inject가 있다.

- **@Autowired**

Spring Framework에서 지원하는 Dependency 정의 용도의 Annotation으로, Spring Framework에 종속적이긴 하지만 정밀한 Dependency Injection이 필요한 경우에 유용하다.

- **@Resource**

JSR-250 표준 Annotation으로 Spring Framework 2.5.* 부터 지원하는 Annotation이다. @Resource는 JNDI 리소스(datasource, java messaging service destination or environment entry)와 연관지어 생각할 수 있으며, 특정 Bean이 JNDI 리소스에 대한 Injection을 필요로 하는 경우에는 @Resource를 사용할 것을 권장한다.

- **@Inject**

JSR-330 표준 Annotation으로 Spring 3 부터 지원하는 Annotation이다. 특정 Framework에 종속되지 않은 어플리케이션을 구성하기 위해서는 @Inject를 사용할 것을 권장한다. @Inject를 사용하기 위해서는 클래스 패스 내에 JSR-330 라이브러리인 javax.inject-x.x.jar 파일이 추가되어야 함에 유의해야 한다.

@Autowired, @Resource, @Inject를 사용할 수 있는 위치는 다음과 같이 약간의 차이가 있으므로 필요에 따라 적절히 사용하면 된다.

- @Autowired : 멤버변수, setter 메소드, 생성자, 일반 메소드에 적용 가능
- @Resource : 멤버변수, setter 메소드에 적용가능
- @Inject : 멤버변수, setter 메소드, 생성자, 일반 메소드에 적용 가능

@Autowired, @Resource, @Inject를 멤버변수에 직접 정의하는 경우 별도 setter 메소드는 정의하지 않아도 된다.

2.2.1.@Inject

Spring의 @Autowired와 동일한 역할을 수행하는 표준 Annotation이다. 단, @Autowired와 달리 'required' 속성을 가지고 있지 않다. 또한 @Named와 같이 사용하였을 경우 정의된 Bean 이름을 이용하여 Injection이 수행된다. 다음은Core Plugin 설치로 추가된 서비스 클래스 ~/moviefinder/service/impl/MovieServiceImpl.java의 일부로써 @Inject를 사용한 예이다.

```
@Service("movieServiceImpl")
@Transactional(rollbackFor = { Exception.class }, propagation = Propagation.REQUIRED)
public class MovieServiceImpl implements MovieService {
```

```

@Inject
@Named("coreMovieDao")
private MovieDao movieDao;

// ...
}

```

2.2.2. @Autowired

@Autowired는 Spring에 종속적이긴 하지만, 적용할 수 있는 위치가 @Resource나 @Inject보다 다양하고, 정밀한 Dependency Injection이 필요한 경우에 유용하다.

다음은 @Autowired를 사용한 예이다.

```

@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Autowired
    MovieDao movieDao;
}

```

@Autowired 적용 위치 별로 사용 예를 들면 다음과 같다.

- 생성자 및 멤버 변수

```

@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Autowired
    MovieDao movieDao;
    MessageSource messageSource;

    @Autowired
    public MovieServiceImpl(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
}

```

위의 예제와 같이 @Autowired를 사용하면 MovieServiceImpl 클래스가 생성될 때 Spring Container에 의해서 MessageSource 타입의 Bean이 생성자의 argument로 자동으로 injection 된다. 또한 movieDao 멤버변수에도 @Autowired가 적용되어 있으므로 MovieDao 타입의 Bean이 자동 injection된다.

- setter 메소드

```

@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    MovieDao movieDao;
    @Autowired
    public void setMovieDao(MovieDao movieDao) {
        this.movieDao = movieDao;
    }
}

```

Spring Container에 의해서 자동으로 setMovieDao() 메소드가 호출되어 MovieDao 타입의 Bean이 movieDao 멤버변수로 injection된다.

- 일반 메소드

```

@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    MovieDao movieDao;
    MessageSource messageSource;
}

```

```

@Autowired
public void prepare(MovieDao movieDao, MessageSource messageSource) {
    this.movieDao = movieDao;
    this.messageSource = messageSource;
}
}

```

@Resource 와는 달리 위의 prepare()와 같은 일반 메소드에도 @Autowired를 적용함으로써 Spring Container에 의한 Dependency Injection 처리를 할 수 있다. 위의 예제에서는 MovieDao 타입의 Bean이 movieDao로, MessageSource 타입의 Bean이 messageSource로 injection된다.

- 배열이나 Collection 형태의 멤버변수와 메소드

```

@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    MovieDao movieDao;
    @Autowired
    Genre[] genres;
}

```

```

@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    MovieDao movieDao;
    Set<Genre> genres;
    @Autowired
    public void setGenres(MovieDao movieDao, Set<Genre> genres) {
        this.movieDao = movieDao;
        this.genres = genres;
    }
}

```

위 예제 소스의 경우, Spring Container에 등록된 Genre 타입의 Bean들이 모두 genres 배열 (또는 collection)에 injection된다.

- Map(key=Bean Name, value=Bean 객체) 형태의 멤버변수와 메소드

```

@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    MovieDao movieDao;
    Map<String, Genre> genres;
    @Autowired
    public void setGenre(MovieDao movieDao, Map<String, Genre> genres) {
        this.movieDao = movieDao;
        this.genres = genres;
    }
}

```

위 예제 소스의 경우, Spring Container에 등록된 Genre 타입의 Bean들이 Bean name이 key로, Bean 객체가 value인 쌍으로 모두 genres Map에 injection된다.

기본적으로 @Autowired가 적용된 참조 관계는 반드시 해당 빈이 존재해야 하지만, **required 속성을 false로 설정하는 경우에는** 해당되는 Bean을 찾지 못하더라도 에러가 발생하지 않는다.

```

@Service
public class MovieServiceImpl implements MovieService {
    @Autowired(required=false)
    private MovieDao movieDao;
}

```

또한, @Resource에서 설명했던 바와 같이 @Autowired도 BeanFactory, ApplicationContext, ResourceLoader, ApplicationEventPublisher, MessageSource 인터페이스와 하위 인터페이스들을 별도 설정 없이 바로 사용할 수 있게 해준다.

```
@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Autowired
    ApplicationContext context;
}
```

2.2.3. @Resource

@Resource는 Bean name을 지정하여 Dependency Injection을 하고자 하는 경우에 사용한다. @Resource는 name이라는 속성을 가지고 있어서, Spring Container가 @Resource로 정의된 요소에 injection하기 위한 Bean을 검색할 때, name 속성에 지정한 이름을 검색할 Bean Name으로 사용한다.

```
@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Resource
    MessageSource messageSource;
    @Resource (name="movieDao")
    MovieDao movieDao;
}
```

명시적으로 name 속성에 이름을 지정하지 않는 경우, 검색할 Bean Name은 다음과 같은 규칙을 따른다.

- @Resource가 멤버 변수에 정의되었을 때 : 멤버 변수의 이름
- @Resource가 setter 메소드에 정의되었을 때 : 해당 setter 메소드의 이름에서 'set'을 제외한 이름 (첫 글자는 소문자)

예) setFoo(...) -> 'foo'

해당하는 Bean Name으로 injection할 Bean을 찾지 못했을 경우에는 @Autowired 처럼 Bean의 type으로 검색한다.

@Resource를 이용하면 BeanFactory, ApplicationContext, ResourceLoader, ApplicationEventPublisher, MessageSource 인터페이스와 하위 인터페이스들을 별도 설정 없이 바로 사용 가능하다.

```
@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Resource
    ApplicationContext context;
}
```

2.2.4. @Qualifier

type-driven injection의 경우 Spring Container가 해당 Bean을 찾을 때 객체의 type을 기준으로 검색을 수행하게 된다. 이와 같은 경우 동일한 type의 Bean이 여러 개 검색되었을 때 injection 대상이 되는 Bean을 결정하기 위한 세밀한 제어가 필요하며 이 때 @Qualifier를 사용할 수 있다.

2.2.4.1. Spring @Qualifier

기본적으로 @Autowired는 type-driven injection 형태로 동작하여, 동일한 객체 type의 Bean이 여러 개 검색되었을 때 injection 대상이 되는 Bean을 결정하기 위해 @Qualifier를 사용할 수 있다.

다음은 @Autowired와 함께 @Qualifier를 사용한 예이다.

```
@Service("movieServiceImpl")
public class MovieServiceImpl implements MovieService {
    @Autowired
    @Qualifier
    MessageSource messageSource;
```

```
@Qualifier("sports")
Movie sportsMovie;
}
```

위와 같이 정의하면 "sports"라는 qualifier 속성 값이 정의된 Bean이 sportsMovie 멤버변수로 injection 된다.

위의 @Qualifier에 의해 연결될 Bean은 다음과 같이 정의할 수 있다.

```
<?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-4.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.0.xsd">

    <context:annotation-config/>

    <bean class="org.anyframe.sample.di.qualifier.moviefinder.domain.Movie">
        <qualifier value="sportsMovie"/>
        <!-- inject any dependencies required by this bean -->
    </bean>
    <bean class="org.anyframe.sample.di.qualifier.moviefinder.domain.Movie">
        <qualifier value="livingMovie"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieServiceImpl"
          class="org.anyframe.sample.di.qualifier.moviefinder.service.MovieServiceImpl"/>
</beans>
```

2.2.4.2.JSR-330 @Qualifier

JSR-330 @Qualifier는 앞서 언급한 Spring @Qualifier 또한 type driven injection 수행시 정밀한 제어를 위해 사용될 수 있다. 단 Spring @Qualifier와 다르게 Qualifier Annotation을 정의하는데만 적용될 수 있다. 다음은 javax.inject.Qualifier를 사용하여 정의된 @Qualifier의 예로 'type'이라는 속성을 가지고 있다.

```
//...
import javax.inject.Qualifier;

@Target( { ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DaoQualifier {
    public abstract String type() default "query";
}
```

MovieDao라는 인터페이스가 2개의 구현체(MovieDaoQueryImpl, MovieDaoHibernateImpl)를 가지고 있다고 가정해보자. 다른 Bean에서 MovieDao의 구현체들에 대해 Type Injection할 수 있도록 하기 위해 해당 Bean을 정의할 때 @Named와 함께 @DaoQualifier를 사용할 수 있을 것이다.

```
@Named
@DaoQualifier(type = "hibernate")
public class MovieDaoHibernateImpl implements MovieDao {
    // ...
}

@Named
```

```
@DaoQualifier
public class MovieDaoQueryImpl implements MovieDao {
    // ...
}
```

앞서 정의한 Bean을 Injection하기 위해서는 다음과 같이 @Inject와 함께 @DaoQualifier를 사용하면 된다.

```
@Named
public class MovieServiceImpl implements MovieService {

    @Inject
    @DaoQualifier(type = "hibernate")
    private MovieDao hibernateMovieDao;

    @Inject
    @DaoQualifier
    private MovieDao queryMovieDao;

    // ...
}
```

위에서 언급한 JSR-330 qualifier 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-di-qualifier를 통해 다운로드받을 수 있다.

2.2.5.Provider

JSR-303에서 제공하는 인터페이스로, 참조하고자 하는 Bean을 직접 Inject하지 않고 Inject 대상이 되는 클래스 타입 T에 대해 Provider<T> 형태로 Inject한다. javax.inject.Provider를 통해 Injection을 수행하는 경우 Spring에서는 DefaultListableBeanFactory 내의 DependencyProvider라는 구현체의 get() 메소드를 이용하여 Generic Type으로 제공된 T 타입의 새로운 인스턴스를 전달해줄도록 하고 있다. 따라서, Singleton Bean에서 Prototype Bean을 참조하고자 할 때 적용할 수 있다.

다음은 Provider<T> 형태로 특정 Bean을 참조한 예이다.

```
@Named
public class MovieServiceImpl implements MovieService {
    @Inject
    private Provider<MovieDao> movieDaoFactory;

    public Movie get(String movieId) throws Exception {
        // get movieDao instance calling get()
        return movieDaoFactory.get().get(movieId);
    }
}
```

위의 코드에서는 Provider 타입의 movieDaoFactory 객체를 통해 get() 메소드를 호출할 때마다 새로운 MovieDao 인스턴스를 전달받게 될 것이다.

특정 Bean을 참조하는 경우 직접 Inject하지 않고 Provider<T> 형태로 Inject하였을 때 다음과 같은 이점을 제공한다. (출처: JSR-330 Dependency Injection for Java 1.0 Final Release for Documentation)

- retrieving multiple instances
- lazy or optional retrieval of an instance
- breaking circular dependencies
- abstracting scope so you can look up an instance in a smaller scope from an instance in a containing scope

위에서 언급한 JSR-330 provider 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-di-provider를 통해 다운로드받을 수 있다.

2.2.6.Generic Types as a form of Qualifier

Spring 4에서는 @Qualifier 이외에도 Generic Type을 이용해 Autowiring을 수행하는 기능을 제공한다. 이 기능을 통해 좀 더 유연한 형태로 코드를 구성할 수 있다.

다음은 Generic Type을 이용해 특정 Bean을 참조한 예이다.

```
@Service("genreService")
public class GenreServiceImpl implements GenreService {

    @Autowired
    private GenericDao<Genre> genreDao;

    public List<Genre> getList() throws Exception {
        return genreDao.getList();
    }
}

public interface GenericDao<T> {

    public void setJdbcDaoDataSource(DataSource dataSource);

    public List<T> getList();

    ...
}

@Repository("genreDao")
public class GenreDao extends JdbcDaoSupport implements GenericDao<Genre> {

    @Inject
    public void setJdbcDaoDataSource(DataSource dataSource) {
        super.setDataSource(dataSource);
    }

    public List<Genre> getList() {
        String sql = "SELECT GENRE_ID, NAME FROM GENRE ORDER BY NAME";
        return super.getJdbcTemplate().query(sql, new BeanPropertyRowMapper<Genre>(Genre.class));
    }

    ...
}
```

위의 코드에서 보듯이 Generic Qualifier의 사용 형태에 따라 기존에 Provider를 이용한 코드 구성을 대체할 수 있다.

Generic Qualifier를 사용하는 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-genericqualifier를 통해 다운로드받을 수 있다.

2.2.7.Ordered Autowiring in Arrays

Spring 4에서는 List나 Array type에 대한 Injection 시에 Element에 대한 정렬을 위해 @Ordered Annotation을 사용할 수 있다.

다음의 예제를 보자.

```

@Service("movieService")
public class MovieServiceImpl implements MovieService {

    // All Daos are injected with @Order value...
    @Autowired
    private List<AbstractDao> daos;

    public void showMyDaos() {
        for (AbstractDao dao : daos) {
            ...
        }
    }
}

@Repository
@Order(value = 2)
public class MovieDao implements AbstractDao {

    public String getDaoName() {
        return getClass().getName();
    }
}

@Repository
@Order(value = 1)
public class MovieFinderDao implements AbstractDao {

    public String getDaoName() {
        return getClass().getName();
    }
}

```

만약에 @Ordered 정보가 없다면, 두 DAO가 어떤 순서로 List에 담길지는 알 수 없다. 덧붙이자면, @Ordered Annotation이 아닌 Ordered Interface를 이용하여 같은 구현이 가능하다.

Ordered Autowiring을 사용하는 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-orderedautowiring를 통해 다운로드받을 수 있다.

2.2.8. @Inject / @Autowired / @Resource 비교

Annotation	@Inject	@Autowired	@Resource
Injection 방식	type-driven injection	type-driven injection	name-matching injection
사용가능한 위치	멤버변수, setter 메소드, 생성자, 일반 메소드	멤버변수, setter 메소드, 생성자, 일반 메소드	멤버변수, setter 메소드

2.2.9. Inject 시점에 @Lazy 사용

기존에는 @Component 나 @Bean 객체에 사용하여 해당 객체를 사용시점에 initialize 하고자할 때 사용되었다. 하지만 Spring 4.0 에서 추가된 내용으로, lazy-init 으로 설정된 Bean을 다른 Bean 객체에서

@Autowired나 @Inject를 통해 Injection하고자 할때 @Lazy를 명시함으로써 해당 Bean객체를 실제 사용된 시점에 initialize 할 수 있다.

```
//...
@Repository
@Lazy
public class MovieLazyDao {

    private long time;

    public MovieLazyDao() {
        System.out.println("class MovieLazyDao initialized");
        time = new Date().getTime();
    }

    public void print()
    {
        System.out.println("--> print MovieLazyDao !!! : " + time);
    }
}
```

위와 같이 Lazy-init 되도록 선언된 Bean 객체가 있다고 하자. 기존에 @Autowired 를 사용하여 다른 Bean 객체에서 Injection하여 사용하였을때는 Bean 객체 생성시에 initialize 가 된다.

```
//...
@Service
public class LazyMovieAutowiredLazyServiceImpl implements LazyMovieAutowiredLazyService {

    @Autowired @Lazy
    private MovieLazyDao movieLazyDao;

    /**
     * Lazy 톰 선언된 Component 의 경우 inject 시점에 lazy-init 을 적용 가능함다.
     */
    @Override
    public void testAutowiredLazy() {
        System.out.println("==== Autowired Lazy Test =====");
        System.out.println("call movieLazyDao.print()");
        movieLazyDao.print();
    }
}

//====> 결과
//==== Autowired Lazy Test =====
//call movieLazyDao.print()
//class MovieLazyDao initialized
//--> print MovieLazyDao !!! : 1393495717313
```

하지만 위와 같이 @Autowired @Lazy 로 객체를 Injection하였을 때는 Bean 객체 생성시에 initialize 되지 않고, Bean 객체가 실제로 사용되는 movieLazyDao.print() 시에 initialize 됨을 확인할 수가 있다.



@Scope(value="prototype") 으로 선언된 Bean 객체 @Lazy 사용 Injection 시 주의사항

@Scope(value="prototype") 으로 선언된 Bean의 경우 객체를 사용할 때마다 Bean 객체를 생성한다는 특징이 있다. 실제로 해당 Bean 객체를 Injection 하게 될 경우에는 Injection한 만큼 Bean 객체가 생성이 된다. prototype 의 scope 을 가지는 Bean 의 경우도 위의 @Lazy

Annotation 을 통하여 Bean 객체의 사용 시점에 Bean 객체를 생성할 수가 있다. 단, 실행 시점마다 Bean 객체가 새로 생성되므로, 용도에 맞게 주의하여 사용해야 한다.

```
//...
@Repository
@Scope(value="prototype")
public class MoviePrototypeDao {

    private long time;

    public MoviePrototypeDao() {
        System.out.println("class MoviePrototypeDao initialized");
        time = new Date().getTime();
    }

    public void print()
    {
        System.out.println("--> print MoviePrototypeDao !!! : " + time);
    }
}
```

```
//...
@Service
public class PrototypeMovieAutowiredLazyServiceImpl implements
    PrototypeMovieAutowiredLazyService {

    @Autowired @Lazy
    private MoviePrototypeDao moviePrototypeDao;

    /**
     * scope=prototype Component 의 경우 inject 시점에 lazy-init 을 적용 가능하나
     * 실행시마다 Bean 객체가 생성된다.
     */
    @Override
    public void testAutowiredLazy() {
        System.out.println("==== Prototype Movie in Autowired Lazy Test =====");
        System.out.println("call moviePrototypeDao.print()");
        moviePrototypeDao.print();
        System.out.println("call moviePrototypeDao.print()");
        moviePrototypeDao.print();
    }
}

//====> 결과
//==== Prototype Movie in Autowired Lazy Test =====
//call moviePrototypeDao.print()
//class MoviePrototypeDao initialized
//--> print MoviePrototypeDao !!! : 1393495675601
//call moviePrototypeDao.print()
//class MoviePrototypeDao initialized
//--> print MoviePrototypeDao !!! : 1393495675619
```



Injection 시에 @Lazy 설정시 유의사항

Injection 시에 사용되는 Annotation 으로는 @Inject / @Autowired / @Resource 세가지가 있으나 @Lazy 를 통해서 lazy-init 이 설정가능한 Annotation 은 @Inject / @Autowired 두가지임을 염두에 두도록 한다.

위의 injection 시점에서의 @Lazy사용에 관한 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-lazyinit를 통해 다운로드받을 수 있다.

2.3.LifeCycle Annotation



IoC의 Life Cycle 에서 설명한 바와 같이 Bean의 LifeCycle은 Initializaion ->Activation -> Destruction으로 구성되어 있으며, LifeCycle 메소드를 정의하는 경우 컨테이너 기동시 또는 종료시 필요한 로직을 수행할 수 있게 된다. Bean을 초기화 또는 소멸화 하는 시점에 별도 작업이 필요한 경우 기존에는 InitializingBean과 DesposableBean 인터페이스를 상속하거나, Bean 정의시 명시적으로 초기화 메소드나 소멸화 메소드를 별도로 지정해야 했다. 그러나, 다음과 같은 Annotation을 사용하면 XML 정의 또는 별도 인터페이스 상속없이 Bean의 LifeCycle 관리가 가능해진다.

2.3.1.@PostConstruct

JSR-250 표준 Annotation으로 Bean 초기화시 필요한 작업을 담은 메소드에 대해 정의한다. @PostConstruct를 사용하기 위해서는 클래스패스 내에 jsr250-api.jar 파일이 추가되어 있어야 한다.

```

@PostConstruct
// 메소드명은 자유롭게 정의할 수 있다.
public void initialize() {
    // ...
}

```

2.3.2.@PreDestroy

JSR-250 표준 Annotation으로 Bean 소멸시 필요한 작업을 담은 메소드에 대해 정의한다. @PreDestroy를 사용하기 위해서는 클래스패스 내에 jsr250-api.jar 파일이 추가되어 있어야 한다.

```

@PreDestroy
// 메소드명은 자유롭게 정의할 수 있다.
public void dispose() {
    // ...
}

```

2.3.3. Combining lifecycle mechanisms

앞에서 설명한 바와 같이, Spring 2.5에서 bean lifecycle을 관리할 수 있는 방법은 다음과 같이 세가지가 있다.

- InitializingBean과 DisposableBean callback 인터페이스 이용
- 사용자가 작성한 초기화/소멸화 메소드를 XML에서 init-method/destroy-method 속성을 이용하여 정의
- @PostConstruct와 @PreDestroy annotation 이용

위의 3가지 방법이 동시에 존재하는 경우(예를 들어, 3가지 방법이 각각 정의된 클래스가 Parent-child 관계를 가지는 경우), 실행되는 순서는 다음과 같다.

Initialization 메소드

1. @PostConstruct를 이용하여 정의한 메소드
2. InitializingBean 인터페이스의 afterPropertiesSet() 메소드
3. XML에서 init-method 속성으로 정의된 초기화 메소드

Destroy 메소드

1. @PreDestroy를 이용하여 정의한 메소드
2. DisposableBean 인터페이스의 destroy() 메소드
3. XML에서 destroy-method 속성으로 정의된 소멸화 메소드

2.4. Meta-Annotation

Spring에서는 Meta-Annotation을 이용하여 Custom Annotation을 직접 생성하여 사용할 수 있다. (예를 들어, Spring 4에서 새롭게 등장한 @RestController도 @Controller와 @ResponseBody를 조합하여 새로운 Stereo Type Annotation을 만든 것이다.) 이러한 Custom Annotation 생성을 통해 새로운 의미 및 특성을 모듈에 반영하는 것이 가능하다.

다음은 Meta-Annotation을 활용해 Custom Annotation을 생성한 예이다.

```
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repository()
public @interface Dao {

    String value() default "dao";

    boolean isDatabase() default true;

}

@Dao("movieDao")
public class MovieDao extends JdbcDaoSupport {

    ...

}
```

위의 예제는 단순하게 Custom Annotation을 설명하기 위해 만든 코드이므로 큰 의미가 없어 보이지만, 상황에 따라서는 유용하게 쓰일 수 있다. (위의 예제에서는 isDatabase()를 이용해 DBMS용 Repository 인지 아닌지를 판별)

2.5.Resources

- 다운로드

다음에서 테스트 DB를 포함하고 있는 hsqldb.zip과 sample 코드를 포함하고 있는 anyframe-sample-annotation.zip 파일을 다운받은 후, 압축을 해제한다. 그리고 hsqldb 폴더 내의 start.cmd (or start.sh) 파일을 실행시켜 테스트 DB를 시작시켜 놓는다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후 mvn clean jetty:run이라는 명령어를 실행시킨다. Jetty Server가 정상적으로 시작되었으면 브라우저를 열고 주소창에 <http://localhost:8080/> anyframe-sample-annotation를 입력하여 실행 결과를 확인한다.

- Eclipse 기반 실행 - m2eclipse, WTP 활용

Eclipse에서 압축 해제 프로젝트를 import한 후, 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭하고 컨텍스트 메뉴에서 Maven > Enable Dependency Management를 선택하여 컴파일 에러를 해결한다. 그리고 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭한 후, 컨텍스트 메뉴에서 Run As > Run on Server (Tomcat 기반)를 클릭한다. Tomcat Server가 정상적으로 시작되었으면 브라우저를 열고 주소창에 <http://localhost:8080/>anyframe-sample-annotation를 입력하여 실행 결과를 확인한다.

- Eclipse 기반 실행 - WTP 활용

Eclipse에서 압축 해제 프로젝트를 import한 후, build.xml 파일을 실행하여 참조 라이브러리를 src/main/webapp 폴더의 WEB-INF/lib내로 복사시킨다. 해당 프로젝트를 선택하고 마우스 오른쪽 버튼을 클릭한 후, 컨텍스트 메뉴에서 Run As > Run on Server를 클릭한다. Tomcat Server가 정상적으로 시작되었으면 브라우저를 열고 주소창에 <http://localhost:8080/>anyframe-sample-annotation를 입력하여 실행 결과를 확인한다. (* build.xml 파일 실행을 위해서는 \${ANT_HOME}/lib 내에 maven-ant-tasks-2.0.10.jar 파일이 있어야 한다.)

Name	Download
maven-ant-tasks-2.0.10.jar	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/maven-ant-tasks-2.0.10.jar]
anyframe-sample-di-qualifier.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-di-qualifier.zip]
anyframe-sample-di-provider.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-di-provider.zip]
anyframe-sample-genericqualifier.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/]

Name	Download
	core/1.6.0/reference/sample/ anyframe-sample-genericqualifier.zip]
anyframe-sample-metaannotation.zip	Download [http:// dev.anyframejava.org/docs/ anyframe/plugin/essential/ core/1.6.0/reference/sample/ anyframe-sample-metaannotation.zip]
anyframe-sample-orderedautowiring.zip	Download [http:// dev.anyframejava.org/docs/ anyframe/plugin/essential/ core/1.6.0/reference/ sample/anyframe-sample- orderedautowiring.zip]
anyframe-sample-lazyinit.zip	Download [http:// dev.anyframejava.org/docs/ anyframe/plugin/essential/ core/1.6.0/reference/sample/ anyframe-sample-lazyinit.zip]

3.Bean Definition Profiles

실제 상황에서 필요한 가장 빈번한 요구사항은 흔히 개발과 운영으로 구분되는 각각의 환경 하에서 서로 다른 bean 설정을 등록하도록 하는 기능이다. 아마 가장 흔한 사례는 개발 단계에서 일반적인 datasource 를 사용하다가 운영 단계로 배포시 JNDI 기반의 datasource 를 사용하도록 하는 것이다. Spring 3.1 버전부터 제공되는 Bean Definition Profiles 기능은 이런 경우를 해결하기 위한 일반적인 방법을 제시한다.

3.1.XML Profiles

다음은 movieService, movieDao와 함께 standalone 형태의 datasource를 설정한 context-movie.xml 예제이다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="...">

  <bean id="movieService"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    <property name="movieDao" ref="movieDao"/>
  </bean>

  <bean id="movieDao"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
    <property name="driverClass" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:file:./db/sampledb" />
    <property name="username" value="sa" />
  </bean>
</beans>
```

상기 예제에서는 Spring 3.0 버전부터 제공되는 jdbc 네임스페이스를 활용하여 내장 데이터베이스 유형을 간단히 설정하고 있다. 만약 운영단계에서 JNDI 기반의 datasource를 사용해야 한다면 위의 설정은 다음과 같이 변경되어야 한다.

```
<beans ...>
  <bean id="movieService" ... />

  <bean id="movieDao"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

각 설정들은 개별적으로 잘 동작하겠지만, 개발과 운영이 전환되어야 하는 상황에서는 위의 두가지 설정 내용을 어떻게 매끄럽게 전환할 것인지 고려해야 할 것이다. 다양한 해결책이 있겠지만 일반적으로 시스템의 환경 변수를 활용하거나, \${placeholder} 토큰을 활용한 <import/> XML 태그를 사용하여 해결한다.

3.1.1.XML 기반의 Profile 적용

Spring 3.1 버전부터는 <beans/> 태그가 profile 개념을 포함하게 되었다. 예제에서의 설정 파일은 이제 다음과 같이 3개의 설정으로 나뉘어 질 수 있는데, context-datasource-*.xml 파일의 profile 속성을 유의해서 살펴보기 바란다. 아래 설정 파일은 context-movie.xml 설정 파일의 내용이다.

```
<beans ...>
  <bean id="movieService" ... />

  <bean id="movieDao"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao">
    <property name="dataSource" ref="dataSource"/>
  </bean>
</beans>
```

다음은 context-datasource-dev.xml 설정 파일의 내용이다. beans 태그의 profile 속성으로 "dev" 가 정의되어 있다.

```
<beans profile="dev">
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
    <property name="driverClass" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:file:./db/sampledb" />
    <property name="username" value="sa" />
  </bean>
</beans>
```

다음은 context-datasource-production.xml 설정 파일의 내용이다. beans 태그의 profile 속성으로 "production"이 정의되어 있다.

```
<beans profile="production">
  <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

3.1.2.중첩된 beans 태그

Spring 3.1 버전부터는 <beans/> 태그를 동일한 파일 내에 중첩하여 사용할 수 있다. 앞 절에서 설명된 3개의 xml 파일을 하나의 설정 파일로 정의하면 다음과 같다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="...">

  <bean id="movieService"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieServiceImpl">
    <property name="movieDao" ref="movieDao"/>
  </bean>

  <bean id="movieDao"
    class="org.anyframe.plugin.core.moviefinder.service.impl.MovieDao">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <beans profile="dev">
    <bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
      <property name="driverClass" value="org.hsqldb.jdbcDriver" />
      <property name="url" value="jdbc:hsqldb:file:./db/sampledb" />
    </bean>
  </beans>
</beans>
```

```
<property name="username" value="sa" />
</bean>
</beans>

<beans profile="production">
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
</beans>
```

이러한 중첩 <beans/> 태그는 Beans Definition Profiles 용이 아닌 일반적인 용도로도 유용하다. 예를 들어 <beans/> 태그의 default-* 속성들 (default-lazy-init, default-init-method, default-destroy-method 등)을 여러개의 bean 에 한번에 정의하고자 할 때에도 활용할 수 있다.

3.2.Profile 활성화

3.2.1.Programmatic Profile 활성화

이제 코드에서 명시적으로 ApplicationContext를 통하여, 앞서 정의된 profile 을 활성화 시키는 방법을 살펴보기로 하자. 다음 코드는 "dev" profile 을 활성화 시키는 예제이다.

```
GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
ctx.getEnvironment().setActiveProfiles("dev");
ctx.load("classpath:/spring/context-*.xml");
ctx.refresh();
```

위의 코드를 실행시키는 경우 각 설정 파일들은 컨테이너에서 다음과 같이 처리된다.

- context-movie.xml 은 profile 속성이 정의되지 않았으므로 항상 파싱되어 처리된다.
- context-datasource-dev.xml 은 profile="dev"로 설정되어 있으며, 현재 "dev" profile 이 활성화되어 있으므로 파싱되어 처리된다.
- context-datasource-production.xml 은 profile="production"로 설정되어 있으며, 현재 "production" profile 이 활성화되어 있지 않으므로 처리가 생략된다.

3.2.2.선언적인 Profile 활성화

기 정의된 profile은 programmatic한 방식 이외에도 시스템 환경 변수, JVM 시스템 프로퍼티, web.xml 내 서블릿 컨텍스트 파라미터 혹은 JNDI 엔트리 등으로 설정된 spring.profiles.active 속성을 통하여 선언적인 방법을 통하여 활성화시킬 수 있다.

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>spring.profiles.active</param-name>
        <param-value>production</param-value>
    </init-param>
</servlet>
```

3.2.3.다중 Profile 활성화

하나의 Profile 만 활성화시킬 수 있는 것이 아니라, 여러개의 Profile 을 활성화 시킬 수도 있다. 다음은 programmatic 한 예제이다.

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

선언적인 방법으로는, spring.profiles.active 값은 콤마로 구분된 profile 명을 리스트 형태로 가질 수 있다.

```
-Dspring.profiles.active="profile1,profile2"
```

다음과 같이 Bean 정의시 하나 이상의 profile 을 표기할 수도 있다.

```
<beans profile="profile1,profile2">
    ...
</beans>
```

3.2.4.@ActiveProfiles

Annotation을 활용하여 profile을 활성화시킬 수 있다. 다음코드는 "Prod" profile을 활성화시키는 예제이다.

```
@ActiveProfiles("Prod")
public class ProdProfileTestWithAnnotation extends
    DefaultProfileTestWithAnnotation {

    @Autowired
    private Movie movie;

    @Test
    @Override
    public void getMovie() {
        assertNotNull(movie);
        assertEquals("Prod", movie.getTitle());
    }

}
```

3.2.5.ActiveProfilesResolver

@ActiveProfile의 속성으로 resolver를 등록하여 사용자가 직접 programmatic한 방식으로 활성화시킬 수 있다. 다음코드는 MovieActiveProfilesResolver를 resolver로 지정한 예제이다.

```
@ActiveProfiles(resolver = MovieActiveProfilesResolver.class)
public class DevProfileTestWithAnnotation extends
    DefaultProfileTestWithAnnotation {

    @Autowired
    private Movie movie;

    @Test
    @Override
    public void getMovie() {
        assertNotNull(movie);
        assertEquals("Dev", movie.getTitle());
    }

}
```

org.springframework.test.context.ActiveProfilesResolver Interface의 resolve 메소드를 구현한다 다음 코드는 @ActiveProfile과 유사하게 동작하는 예제이다.

```
public class MovieActiveProfileResolver implements ActiveProfilesResolver{

    @Override
    public String[] resolve(Class<?> testClass) {
        String profile = null;

        if(testClass.getName().contains("Dev")){
            profile = "Dev";
        }else if (testClass.getName().contains("Prod")){
            profile = "Prod";
        }

        return new String[] {profile};
    }

}
```

3.3.@Profile

Bean Definition Profiles 정의시 Annotation 을 활용하여 정의할 수도 있는데, 이때 사용되는 것이 @Profile 태그이다. 이해를 위하여 다음장에 이어서 나오는 Java Based Configuration 의 내용을 함께 참고하는 것이 좋다.

3.3.1.Annotation 기반의 Profile 적용

XML 기반의 Profile 적용에서 언급되었던 XML 설정 파일의 내용을 Annotation 기반의 Java Based Configuration 으로 변환하면 다음과 같다.

```
@Configuration
@Profile("dev")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new SimpleDriverDataSource(
            new org.hsqldb.jdbcDriver(),
            "jdbc:hsqldb:file:./db/sampledb",
            "sa",
            "");
    }

}
```

```
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }

}
```

```
}
```

동일한 방법으로 XML 기반의 MovieService 설정은 다음과 같이 Java Based Configuratoion 으로 변경될 수 있다.

```
@Configuration
public class MovieServiceConfig {

    @Autowired
    DataSource dataSource;

    @Bean
    public MovieService movieService() {
        return new MovieService(movieDao());
    }

    @Bean
    public MovieDao movieDao() {
        return new MovieDao(dataSource);
    }
}
```

이 때 @AutoWired 를 통해 인젝션되는 dataSource 는 다음과 같이 활성화된 profile 에 따라 결정된다.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("dev");
ctx.scan("org.anyframe.plugin.core.moviefinder");
ctx.refresh();
```

3.4.@Conditional

Spring 4에서부터 @Conditional Annotation을 이용해 원하는 조건의 Bean을 Filtering할 수 있다. 이는 @Profile과 비슷한 기능으로 사용자가 직접 programmatic한 방식으로 서로 다른 bean 설정을 등록할 수도 있다.

3.4.1.Annotation 기반의 Conditional 적용

사용법은 기존 @Profile과 동일하며 Conditional.class를 속성으로 사용한다. 이해를 위하여 다음에 나오는 Java Based Configuration 의 내용을 참고하는 것이 좋다.

```
@Configuration
public class ConditionalConfig {

    @Conditional(DevCondition.class)
    @Bean(name="movie")
    public Movie getDevMovie(){
        Movie movie = new Movie();
        movie.setTitle("Dev");
        return movie;
    }

    @Conditional(ProdCondition.class)
    @Bean(name="movie")
    public Movie getProdMovie(){
        Movie movie = new Movie();
        movie.setTitle("Prod");
        return movie;
    }
}
```

```

}

@Conditional(DefaultCondition.class)
@Bean(name="movie")
public Movie getDefaultMovie(){
    Movie movie = new Movie();
    movie.setTitle("Default");
    return movie;
}
}

```

org.springframework.context.annotation.Condition Interface의 matches 메소드를 구현한다. 다음 코드는 "os.name" 환경변수에 따라 동작하는 Condition 예제이다.

```

public class DevCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata arg1) {
        return context.getEnvironment().getProperty("os.name").contains("Linux");
    }

}

```

```

public class ProdCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata arg1) {
        return context.getEnvironment().getProperty("os.name").contains("Windows");
    }

}

```

```

public class DefaultCondition implements Condition {

    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata arg1) {
        return !context.getEnvironment().getProperty("os.name").contains("Linux") &&
            !context.getEnvironment().getProperty("os.name").contains("Windows");
    }

}

```

3.5.Environment Abstraction (Environment 추상화)

3.5.1.PropertySources

Spring 3.1부터 제공되는 Environment 추상화 기능은 우선순위 설정이 가능한 PropertySource 에 대하여 통합 검색 옵션을 제공한다.

```

ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsFoo = env.containsProperty("foo");

```

```
System.out.println("Does my environment contain the 'foo' property? " + containsFoo);
```

위의 코드를 살펴보면 'foo' 라는 속성이 현재 환경 내에서 정의되어 있는지를 확인하고 있다. 해당 속성값을 검색하기 위해 Environment 객체는 PropertySource 객체들에 대하여 검색을 수행한다.

PropertySource 란 키-값 쌍으로 구성된 속성들의 추상화를 나타내는데, Spring 의 DefaultEnvironment 는 다음과 같이 두개의 PropertySource 로 구성되어 있다.

- JVM 시스템 속성값의 집합 (System.getProperties())와 같은 방식)
- 시스템 환경 변수의 집합 (System.getenv())와 같은 방식)

즉, 'foo' 라는 이름으로 시스템 속성값이 정의되어 있거나, 환경 변수가 정의되어 있다면 env.containsProperty("foo")의 값은 true 가 될 것이다. 검색은 순차적으로 수행되는데, 기본적으로 시스템 속성이 환경 변수보다 우선 순위가 높다. 따라서 env.getProperty("foo")을 실행하게 되면 시스템 속성값이 우선적으로 반환된다. 중요한 점은, 이러한 동작 원리에 대하여 설정 변경이 가능하다는 것이다. 만약 사용자 정의 PropertySource 를 만들었다면 순차적인 검색에 포함되도록 설정할 수 있다.

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

위의 예제 코드에서는 MyPropertySource 를 가장 높은 우선순위로 검색되도록 추가하는 모습을 나타내고 있다. 만약 원하는 값을 MyPropertySource 에서 찾지 못한다면 다른 PropertySource 에서 우선 순서에 따라 검색하게 된다.

3.5.2.PropertySource 활용

이제 실제 개발시 어플리케이션 상에서 어떻게 Environment 와 PropertySource 를 사용할 수 있는지 살펴해보도록 한다.

1. <import/> 구문 내에서의 \${placeholder} 처리

```
<beans>
  <import resource="spring/${customer}-config.xml"/>
</beans>
```

고객 정보에 따라 각기 다른 설정 파일을 로드해야 하는 경우 <import/> 구문과 placeholder 를 통해 'customer' 값을 결정하게 된다. Spring 3.1 이전에는 placeholder 의 값은 오로지 JVM 시스템 속성이나 시스템 환경 변수에서만 검색되었다. 하지만 Spring 3.1 이후부터 Environment 추상화를 통해 검색 우선 순위를 조정할 수 있을 뿐만 아니라 사용자 정의 PropertySource 를 적절히 활용할 수 있게 되었다.

2. bean 정의내에서의 \${placeholder} 처리

```
<context:property-placeholder location="com/bank/config/datasource.properties"/>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
  <property name="driverClass" value="${database.driver}"/>
  <property name="jdbcUrl" value="${database.url}"/>
  <property name="username" value="${database.username}"/>
  <property name="password" value="${database.password}"/>
</bean>
```

일반적으로 Spring에서 bean 정의시 사용된 \${...} placeholder 를 대체하기 위해서 PropertyPlaceholderConfigurer 나 <context:property-placeholder/>를 사용한다. Spring 3.1 버전부터는 <context:property-placeholder/> 에서 더이상 PropertyPlaceholderConfigurer 를 등록하

지 않고 `PropertySourcesPlaceholderConfigurer` 를 등록하는데, 이 컴포넌트는 기존과 동일하게 `datasource.properties` 파일 내에서 `${database.*}` 속성값을 찾지만 속성값을 찾는데 실패한 경우에는 대안으로써 현재의 `Environment` 객체의 `PropertySource` 를 사용한다. 이러한 개선 이전에는 속성값 검색 실패시, 대안으로 오직 시스템 속성값과 환경 변수만을 사용할 수 있었지만, 이제 개발자 측면에서 더 다양한 `PropertySource` 를 추가적으로 활용할 수 있게 되었다.

3.5.3. 웹 어플리케이션내의 `PropertySource` 활용

상당수의 Spring 기반 어플리케이션은 웹 어플리케이션이며, `ApplicationContext`가 `ContextLoaderListener`에 의해 관리된다. 이제 `PropertySource` 를 제어하기 위하여 Spring 에서 제공하는 `ApplicationContextInitializer` 인터페이스를 구현하고, 서블릿 컨텍스트 파라미터에 `contextInitializerClasses` 값을 정의한 예제를 살펴보기로 한다. `web.xml` 의 일부는 다음과 같다.

```
<context-param>
  <param-name>contextInitializerClasses</param-name>
  <param-value>com.bank.MyInitializer</param-value>
</context-param>
```

`ApplicationContextInitializer`를 구현한 예제 클래스는 다음과 같다.

```
public class MyInitializer implements
  ApplicationContextInitializer<ConfigurableWebApplicationContext> {
  public void initialize(ConfigurableWebApplicationContext ctx) {
    PropertySource ps = new MyPropertySource();
    ctx.getEnvironment().getPropertySources().addFirst(ps);
    // 기타 컨텍스트 초기화 작업 수행 ....
  }
}
```

`ApplicationContextInitializer`를 구현하고 이를 등록함으로써, 간단한 방법으로 웹 어플리케이션의 `ApplicationContext`가 refresh 되기 전에 추가 처리를 할 수 있다.

4. Java based Configuration

Spring 3에서는 Spring Java Configuration 프로젝트 [<http://www.springsource.org/javaconfig>]의 일부 주요 특징들을 추가함으로써 Java 기반의 Configuration 정의가 가능하도록 지원하고 있다. Java 기반의 속성 정의는 Java 코드를 중심으로 이루어지므로 Injection 속성 정의시 Type 오류가 있으면 컴파일부터 수행되지 않으므로 Type Safety를 보장하게 된다. 또한 Bean 인스턴스 관리를 로직으로 직접 구현해주기 때문에 Bean 구현체가 Spring에 의존되지 않고, 순수한 Java 코드로만 구현될 수 있도록 보장해준다.

- @Configuration
- @Bean
- @Lazy
- @DependsOn
- @Primary
- @Value
- @Import
- @ImportResource

활용 가능한 Annotation들은 위에서 나열한 바와 같으며, 본 섹션에서는 이러한 Annotation들에 대해 예제와 함께 자세히 살펴보도록 하자.

Java 기반의 Configuration 정의시 가장 기본이 되는 Annotation은 @Configuration과 @Bean이다. @Configuration은 클래스 레벨에 정의가능한 Annotation이다. @Configuration 정의를 포함한 클래스는 Bean 정의 정보를 담고 있어 Spring Container에 의해 처리되는 Configuration 클래스임을 의미한다. @Bean은 메소드 레벨에 정의 가능한 Annotation으로 XML 기반의 속성 정보 중 <bean/>과 동일한 역할을 수행한다.

```
@Configuration
public class MovieFinderConfig {
    // ...
    @Bean
    public MovieFinder movieFinder() {
        return new MovieFinderImpl(movieDao);
    }
}
```

위 코드에서 언급한 MovieFinderConfig 클래스는 Configuration 클래스로써 'movieFinder'라는 이름을 가진 Bean을 정의하고 있음을 알 수 있다. 위 코드 내용을 XML 형태로 변경해 보면 다음과 같다.

```
<bean id="movieFinder" class="org.anyframe.sample.javaconfig.moviefinder.service.impl">
  <constructor-arg ref="movieDao"/>
</bean>
```

4.1. Bean Management

앞서 언급한 바와 같이 @Bean은 메소드 레벨에 정의 가능한 Annotation으로 특정 Bean을 정의하기 위해 사용한다. XML 기반의 속성 정보 중 <bean/>과 동일한 역할을 수행하며, @Configuration 또는 @Component 클래스 내에 정의 가능하다. @Bean 정의가 추가된 메소드는 해당하는 Bean의 인스턴스 생성하여 전달하는 로직을 포함하고 있어야 하며 기본적으로 Spring Container는 메소드명을 Bean 이름으로 등록한다.

```
@Bean
public MovieFinder movieFinder() {
    return new MovieFinderImpl(movieDao);
}
```

위 코드에 의하면 @Bean 정의가 추가된 movieFinder() 메소드로 인해 'movieFinder'라는 이름의 Bean이 Spring Container에 등록될 것이다. 또한 'movieFinder' Bean을 요청하면 정의된 메소드 로직에 의해 MovieDao 객체가 셋팅된 MovieFinderImpl 객체가 전달될 것이다.

4.1.1.Naming

@Bean Annotation은 'name'이라는 속성 정보를 가지고 있다. name 속성에 대해 값을 부여하는 경우 이 값이 해당 Bean의 이름이 된다.

```
@Bean(name="movieFinderImpl")
public MovieFinder movieFinder() {
    return new MovieFinderImpl(movieDao);
}
```

4.1.2.Lifecycle Management

@Bean을 이용하여 정의된 Bean들에 대해서도 XML이나 Annotation 기반의 Bean들과 동일하게 기본 Lifecycle 관리가 가능하다. 즉, 해당 Bean이 @PreDestroy, @PostConstruct와 같은 JSR-250 Annotation을 포함하고 있거나 Spring의 InitializingBean, DisposableBean 등과 같은 인터페이스를 구현하였을 경우 Spring Container에 의해 해당 Bean의 Lifecycle이 관리된다. 이 외에도 @Bean은 'init-method', 'destroy-method'라는 속성 정보를 가질 수 있어서 속성값을 부여하는 경우 초기화/소멸화시에 정의된 메소드가 실행된다. 이것은 <bean/>의 init-method, destroy-method와 동일한 역할을 수행한다.

```
@Bean(initMethod = "initialize", destroyMethod = "destroy")
public MovieFinder movieFinder() {
    return new MovieFinderImpl(movieDao);
}
```

위 코드에 의하면 'movieFinder'라는 Bean의 초기화 시점에는 MovieFinderImpl.initialize(), 소멸화 시점에는 MovieFinderImpl.destroy() 메소드가 각각 실행될 것이다.

Spring Container는 시작 시점에 모든 Singleton Bean을 미리 로딩함으로써, 그 Bean이 필요할 때 즉시 사용될 수 있도록 보장해준다. 그러나 Container 시작 시점에 특정 Singleton Bean을 인스턴스화시키지 않고 처음으로 해당 Bean에 대해 요청이 들어왔을 때 인스턴스화시키기 위해서는 @Lazy 설정을 부여해 주어야 한다. 이것은 <bean/>의 lazy-init과 동일한 역할을 수행한다.

```
@Bean
@Lazy
public MovieFinder movieFinder() {
    return new MovieFinderImpl(movieDao);
}
```

4.1.3.Scope

@Bean과 함께 @Scope 정의를 추가하는 경우 해당 Bean에 대해 특정 Scope을 부여할 수 있다. @Scope을 부여하지 않는 경우 기본적으로 Singleton Scope이 적용된다.

```
@Bean
@Scope("prototype")
public MovieFinder movieFinder() {
```

```
return new MovieFinderImpl(movieDao);
}
```

또한 request, session, globalSession Scope의 Bean에 대한 요청시 전달될 AOP Proxy 객체를 만들기 위해서 'proxyMode'라는 속성값을 추가적으로 부여할 수 있다. 'proxyMode'는 기본적으로 ScopedProxyMode.NO로 지정되며 ScopedProxyMode.TARGET_CLASS 또는 ScopedProxyMode.INTERFACES으로 정의 가능하다. 이것은 <bean/> 하위의 <aop:scoped-proxy/>와 동일한 역할을 수행한다.

```
@Bean
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public MoviePreferences moviePreferences() {
    return new MoviePreferences();
}

@Bean
public MovieFinder movieFinder() {
    return new MovieFinderImpl(moviePreferences());
}
```

4.1.4. Dependency Injection

Bean 사이에 참조 관계가 성립될 경우 기본적으로 Injection은 참조하려는 Bean에 해당하는 메소드를 호출함으로써 이루어진다.

```
@Configuration
public class MovieFinderConfig {
    @Bean
    public MovieFinder movieFinder() {
        return new MovieFinderImpl(movieDao());
    }

    @Bean
    public MovieDao movieDao() {
        return new MovieDao();
    }
}
```

'movieFinder' Bean이 'movieDao' Bean을 참조하고 있다고 가정해 보자. 이를 Java 기반의 Configuration으로 표현하기 위해서는 위의 코드에서와 같이 movieFinder() 메소드 내에서 MovieFinderImpl 인스턴스 생성시 movieDao()라는 메소드를 호출함으로써 'movieDao' Bean을 Injection할 수 있다. 또는 MovieFinderImpl 객체의 setter를 호출할 때 movieDao() 호출 결과를 전달함으로써 'movieDao' Bean을 Injection할 수도 있을 것이다.

```
@Configuration
public class MovieFinderConfig {
    @Bean
    public MovieFinder movieFinder() {
        MovieFinderImpl movieFinder = new MovieFinderImpl();
        movieFinder.setMovieDao(movieDao());
        return movieFinder;
    }

    @Bean
    public MovieDao movieDao() {
        return new MovieDao();
    }
}
```

참조 대상 Bean이 XML/Annotation 기반으로 정의되었거나 다른 Configuration 클래스에 정의된 경우 Spring에서 Dependency Injection 처리를 위해 지원하는 Annotation(@Inject, @Autowired, @Resource)을 그대로 적용할 수도 있다.

```
@Configuration
public class MovieDaoConfig {

    @Bean
    public MovieDao movieDao() {
        MovieDao movieDao = new MovieDao();
        return movieDao;
    }
}
```

```
@Configuration
@Import(value = { MovieDaoConfig.class })
public class MovieFinderConfig {

    @Autowired
    private MovieDao movieDao;

    @Bean
    public MovieFinder movieFinder() {
        return new MovieFinderImpl(movieDao);
    }
}
```

해당 Bean 이전에 초기화되어야 하는 하나 이상의 Bean을 명시적으로 강제하기 위해서는 @DependsOn을 활용할 수 있다. 이것은 <bean/>의 depends-on와 동일한 역할을 수행한다.

```
@Configuration
public class MovieFinderConfig {

    @Bean
    public MovieService movieService(){
        return new MovieServiceImpl();
    }

    @Bean
    @DependsOn(value = { "movieService" })
    public MovieFinder movieFinder() {
        return new MovieFinderImpl(movieDao());
    }

    // ...
}
```

위 코드에 의하면 @DependsOn 속성 부여에 의해 'movieFinder' Bean이 초기화되기 전에 'movieService' Bean이 초기화 될 것을 짐작할 수 있다.

동일한 Type을 가지는 Bean이 여러개 정의되어 있어서 Type Injection 대상이 되는 Bean이 여러개 식별되었을 경우 @Primary를 부여한 Bean이 우선적으로 Injection 후보가 된다. 이것은 <bean/>의 primary와 동일한 역할을 수행한다.

```
@Configuration
public class MovieDaoConfig {

    @Bean
    public MovieDao defaultMovieDao() {
        return new MovieDaoImpl();
    }

    @Bean
```

```

@Primary
public MovieDao anotherMovieDao() {
    return new AnotherMovieDaoImpl();
}
}

```

위와 같이 Configuration을 정의한 경우 **@Autowired MovieDao movieDao;**와 같은 코드에 의해 Injection되는 Bean은 @Primary 속성을 부여한 'anotherMovieDao' Bean이 될 것이다.

4.1.5.Method Injection

Setter injection과 Constructor injection을 사용할 경우, 기본적으로 Singleton Bean은 참조하는 Bean들을 Singleton 형태로 유지하게 된다. 그런데 Singleton Bean이 Non Singleton Bean(즉, Prototype Bean)과 참조 관계가 있을 경우에는 다음과 같이 처리해야 한다.

1. Singleton Bean의 구현체 내에는 참조하려는 Non Singleton Bean 타입을 리턴하는 abstract 메소드 정의.
2. Singleton Bean의 구현체 내의 비즈니스 메소드에서는 abstract 메소드를 이용해 Non Singleton Bean을 Injection하여 로직 수행.
3. Java 기반 Configuration 정의시 Singleton Bean에 해당하는 메소드 내에서 인스턴스 생성과 함께 앞서 정의한 abstract 메소드 구현 로직 추가. 이 때 abstract 메소드 구현 로직에서는 Non Singleton Bean의 인스턴스 생성하여 리턴.
4. 위와 같은 순서로 처리된 경우 Singleton Bean의 비즈니스 메소드 내에서 abstract 메소드가 호출될 때마다 해당 Bean의 인스턴스가 가진 abstract 메소드 구현 로직에 의해 새로운 Non Singleton Bean의 인스턴스 전달이 가능해짐. 즉, Singleton Bean에서 Non Singleton Bean에 대한 참조가 가능해짐.

다음은 Singleton Bean('movieFinder')에서 Non Singleton Bean('movieDao')에 대한 참조가 이루어질 수 있도록 하기 위해 정의된 Configuration 클래스의 내용이다.

```

@Configuration
public class MovieFinderConfig {
    @Bean
    @Scope("prototype")
    public MovieDao movieDao() {
        return new MovieDaoImpl();
    }

    @Bean
    public MovieFinder movieFinder() {
        return new MovieFinderImpl() {
            protected MovieDao getMovieDao() {
                return movieDao();
            }
        };
    }
}

```

위 Configuration 클래스에서 언급한 MovieFinderImpl 클래스는 다음과 같은 모습을 취할 것이다.

```

public abstract class MovieFinderImpl implements MovieFinder {
    protected abstract MovieDao getMovieDao();

    public List<Movie> getPagingList(Movie movie, int pageIndex)
        throws Exception{
        return getMovieDao().getPagingList(movie, pageIndex);
    }
}

```

4.1.6.Spring Expression Language

Java 기반 Configuration 정의시 @Value와 함께 Spring Expression Language를 정의하면 Expression 처리 결과를 Bean의 인스턴스 생성시 반영하는 것도 가능하다.

```
@Configuration
public class MovieFinderConfig {
    private @Value("${jdbc.url}") String dbUrl;
    private @Value("${jdbc.username}") String userName;
    private @Value("${jdbc.password}") String password;

    @Bean
    public MovieDao movieDao() {
        return new MovieDaoImpl(dbUrl, userName, password);
    }
}
```

4.1.7.Description Annotation

Spring 4.0에서부터는 Java 기반 Configuration 정의 시 @Description Annotation을 이용하여 Bean에 정보를 기록할 수 있다. 주요 용도로는 Debugging 시의 정보 획득 등이 예상된다. 다음은 Configuration 시 Description을 준 예제이다.

```
@Configuration
public class MovieDaoConfig {

    @Bean
    @Description("Provides a MovieFinderDao bean")
    public MovieFinderDao movieFinderDao() {
        MovieFinderDao movieFinderDao = new MovieFinderDao();
        return movieFinderDao;
    }
}
```

4.1.8.Code Equivalents for Spring's XML namespaces

코드 기반으로 Spring의 XML 네임스페이스(<context:component-scan/>, <tx:annotation-driven/> and <mvc:annotation-driven>)에 대응하는 @Enable Annotation이 Spring 3.1 버전부터 추가되었다. 이러한 Annotation은 Spring 3.0부터 도입된 @Configuration 태그와 같이 사용하도록 설계되어 있다.

4.1.8.1.@ComponentScan

@Configuration 태그와 함께 사용하며 <context:component-scan>과 동일한 기능을 수행할 수 있다.

```
@Configuration
@ComponentScan("com.acme.app.services")
public class AppConfig {
    // various @Bean definitions ...
}
```

4.1.8.2.@EnableTransactionManagement

Spring에서 제공하는 Annotation 기반의 트랜잭션 관리 기능을 활성화시킨다. @Configuration 태그와 함께 사용하며, <tx:*>과 동일한 기능을 수행할 수 있다.

```

@Configuration
@EnableTransactionManagement
public class AppConfig {

    @Bean
    public FooRepository fooRepository() {
        // configure and return a class having @Transactional methods
        return new JdbcFooRepository(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        // configure and return the necessary JDBC DataSource
    }

    @Bean
    public PlatformTransactionManager txManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}

```

4.1.8.3. @EnableWebMvc

@Configuration 태그와 함께 사용하여 WebMvcConfigurationSupport 클래스에 정의된 Spring MVC의 설정을 다음과 같이 불러올 수 있다.

```

@Configuration
@EnableWebMvc
@ComponentScan(basePackageClasses = { MyConfiguration.class })
public class MyWebConfiguration {

}

```

WebMvcConfigurer 인터페이스를 구현하거나, WebMvcConfigurerAdapter 기반의 클래스를 확장하고 메소드를 오버라이딩 하여 불러들여진 설정을 다음과 같이 사용자화 할 수 있다.

```

@Configuration
@EnableWebMvc
@ComponentScan(basePackageClasses = { MyConfiguration.class })
public class MyConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addFormatters(FormatterRegistry formatterRegistry) {
        formatterRegistry.addConverter(new MyConverter());
    }

    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        converters.add(new MyHttpMessageConverter());
    }

    // More overridden methods ...
}

```

4.2. Combining Java and XML Configuration

@Import/@ImportResource를 활용하면 XML 또는 다른 @Configuration 클래스에 정의된 Bean 정보를 참조할 수 있게 된다.

4.2.1.Combine Java Configuration

@Import 정의시 다른 @Configuration 클래스를 속성값으로 부여해주면 현재 @Configuration 클래스에서 다른 @Configuration 클래스 내에 정의된 @Bean 정보를 참조할 수 있게 된다. Import 대상이 되는 @Configuration 클래스가 다수일 경우 {} 내에 ','를 식별자로 하여 클래스를 명시해주면 된다. @Import는 <import/>와 동일한 역할을 수행한다.

```
@Configuration
@Import(value = { MovieDaoConfig.class })
public class MovieFinderConfig {
    @Autowired
    private MovieDao movieDao;

    @Bean
    public MovieFinder movieFinder() {
        return new MovieFinderImpl(movieDao);
    }
}

@Configuration
public class MovieDaoConfig {
    // ...

    @Bean
    public MovieDao movieDao() {
        MovieDao movieDao = new MovieDao();
        return movieDao;
    }
}
```

위에서 언급한 @Configuration MovieFinderConfig 클래스는 MovieDaoConfig 클래스를 @Import하고 있어서 이 클래스 내에 정의된 Bean 'movieDao'를 참조할 수 있게 된다. 다른 @Configuration 클래스 내에 정의된 @Bean을 참조하기 위해서는 @Autowired를 사용하고 있음을 알 수 있다. @Inject, @Resource를 사용하는 것 또한 가능하다.

4.2.2.Combine XML Configuration

@ImportResource 정의시 XML 속성 파일의 위치를 속성값으로 부여해주면 현재 @Configuration 클래스에서 XML 내에 정의된 Bean 정보를 참조할 수 있게 된다. Import 대상이 되는 XML 파일이 다수일 경우 @Import와 동일한 형태로 {} 내에 ','를 식별자로 하여 XML 파일명을 명시해주면 된다.

```
@Configuration
public class MovieDaoConfig {

    @Bean
    public MovieDao movieDao() {
        MovieDao movieDao = new MovieDao();
        return movieDao;
    }
}
```

위 코드에서는 다른 XML 내에 정의된 Bean을 참조하기 위해 @Autowired를 사용하고 있음을 알 수 있다. @Inject, @Resource를 사용하는 것 또한 가능하다.

4.3. Instantiating spring container

Spring 3에서는 @Configuration 클래스를 인식하여 정의된 Bean들을 관리할 수 있도록 하기 위해 ApplicationContext의 구현체인 AnnotationConfigApplicationContext를 추가적으로 제공하고 있다. AnnotationConfigApplicationContext는 @Configuration 클래스 외에도 Stereotype Annotation, JSR-330 Annotation에 대해 인식 가능하다. 다음에서는 @Configuration 클래스를 기반으로 Spring Container를 시작시키는 방법에 대해서 살펴해보도록 하자.

4.3.1. AnnotationConfigApplicationContext

XML/Annotation 기반에서 Spring Container를 시작시키기 위해서는 XmlWebApplicationContext, FileSystemXmlApplicationContext, ClassPathXmlApplicationContext와 같은 구현체를 활용했었다.

```
String[] locations = new String[] { "classpath:spring/context-*.xml" };
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(locations,
    false);
context.refresh();
```

그러나 @Configuration 클래스를 인식할 수 있도록 하기 위해서는 AnnotationConfigApplicationContext 구현체를 이용하여 Spring Container를 시작시켜야 한다. 인식해야 할 @Configuration 클래스가 다수일 경우 해당되는 클래스들을 입력 인자의 값으로 정의해주면 된다.

```
AnnotationConfigApplicationContext context = new
    AnnotationConfigApplicationContext(MovieFinderConfig.class, ...);
```

또는 AnnotationConfigApplicationContext의 Default Constructor를 호출하여 인스턴스를 생성한 뒤 인식 대상이 되는 @Configuration 클래스들을 register할 수도 있다.

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.register(MovieFinderConfig.class, ...);
context.register(...);
context.refresh();
```

Spring Container가 Annotation 기반의 Bean을 검색할 수 있게 하기 위해 정의한 <context:component-scan/>과 유사하게 AnnotationConfigApplicationContext을 이용하여 특정 패키지 하위에 대한 scan도 가능하다. 이렇게 하는 경우 해당 패키지 하위에 속한 모든 @Configuration 클래스가 검색되어 Container에 의해 처리된다.

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.scan("org.anyframe.sample");
context.refresh();
```

4.3.2. AnnotationConfigWebApplicationContext

웹어플리케이션에서 @Configuration 클래스를 인식하여 Spring Container를 시작시키기 위해서는 ContextLoaderListener Listener의 속성 정보인 contextClass, contextConfigLocation의 값을 입력해주면 된다. 이 때, contextClass는 AnnotationConfigWebApplicationContext로 정의해주고, 이를 통해 로드될 @Configuration 클래스들을 contextConfigLocation의 속성값으로 부여해주도록 한다.

```
<context-param>
  <param-name>contextClass</param-name>
  <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-
value>
</context-param>
```

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>org.anyframe.samples.moviefinder.basic.config.MovieFinderConfig</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```



Java 기반 Configuration 정의시 유의사항

다음 코드에서는 'movieFinder1', 'movieFinder2' Bean이 'movieDao' Bean을 참조하고 있다. 'movieDao' Bean을 참조하기 위해 movieDao() 메소드를 호출하고 있기 때문에 'movieFinder1', 'movieFinder2' Bean이 참조하는 MovieDao의 인스턴스가 다를 것이라고 기대할 것이다.

```
@Configuration
public class MovieFinderConfig {

    @Bean
    public MovieFinder movieFinder1() {
        return new MovieFinderImpl(movieDao());
    }

    @Bean
    public MovieFinder movieFinder2() {
        return new MovieFinderImpl(movieDao());
    }

    @Bean
    public MovieDao movieDao() {
        return new MovieDao();
    }
}
```

그러나 Spring에서는 초기화시에 CGLIB을 이용하여 모든 @Configuration 클래스에 대해 subclass화하고 subclass 내의 메소드에서는 특정 Bean의 인스턴스를 생성하기 전에 Container가 Caching된 Singleton Bean의 인스턴스가 있는지 체크하도록 처리하고 있기 때문에 'movieFinder1', 'movieFinder2'는 동일한 'movieDao' 인스턴스를 참조하게 된다.

설명한 바와 같이 Spring에서 @Configuration 클래스를 처리하기 위해 CGLIB을 사용하므로 해당 프로젝트에는 CGLIB 라이브러리가 반드시 필요하며, CGLIB을 이용하여 @Configuration 클래스에 대해 subclass화하는 작업을 위해 @Configuration 클래스는 Default Constructor를 반드시 가져야 하고 final로 정의되지 않도록 해야 함에 유의하도록 한다.

4.4.Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, mvn compile exec:java -Dexec.mainClass=...이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, src/main/java 폴더 하위의 Main.java를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 Run As > Java Application을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
anyframe-sample-javaconfig.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-javaconfig.zip]

5.AOP(Asspect Oriented Programming)

다음 내용은 ZDNet Korea의 제휴 매체인 마이크로소프트웨어에 게재된 내용에서 발췌함. 관점지향 프로그래밍(Asspect Oriented Programming, 이하 AOP)은 지금까지의 프로그래밍 기술 변화의 흐름에 다른 차원의 관점을 제시함으로써 새로운 프로그래밍 패러다임을 이끌어내고 있다고 볼 수 있다. AOP의 필요성을 이해하는데 기초가 되는 개념은 Separation of Concerns로, 다음과 같이 거의 모든 프로그래밍 패러다임은 바로 이 Separation of Concerns 과정을 통해 문제 영역을 독립적인 모듈로 분해한다.

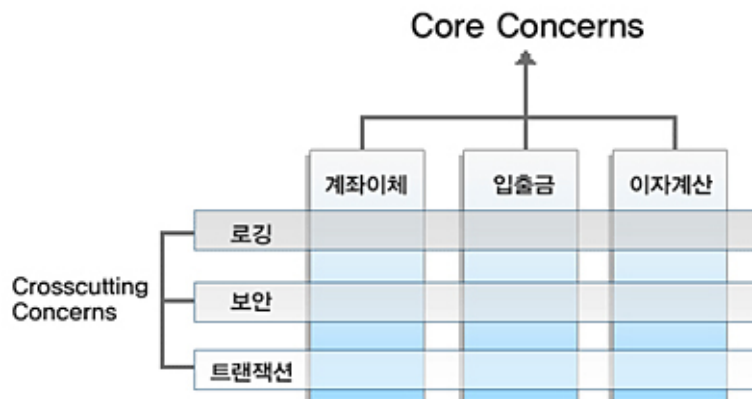
- 절차적 프로그래밍 : 분리된 관심을 프로시저로 구성
- 객체지향 프로그래밍(Object Oriented Programming, 이하 OOP) : 분리된 관심을 클래스로 작성

AOP 필요성

AOP는 OOP를 적용한다고 할지라도 결코 쉽게 분리된 모듈로 작성하기 힘든 요구사항이 실제 어플리케이션 설계와 개발에서 자주 발견된다는 문제 제기에서 출발한다. AOP에서는 이를 Crosscutting Concerns(횡단 관심)라고 한다. 또한 해당 시스템의 핵심 가치와 목적이 그대로 드러난 관심 영역을 Core Concerns(핵심 관심)라고 부른다. 이 Core Concerns는 기존의 객체지향 분석/설계(OOAD)를 통해 쉽게 모듈화와 추상화가 가능하지만 Crosscutting Concerns는 객체지향의 기본 원칙을 지키면서 이를 분리해서 모듈화하는 것이 매우 어렵다.

예를 들어, 은행 업무를 처리하는 시스템을 생각해 보면 Core Concerns는 예금입출금, 계좌잔이체, 이자계산, 대출처리 등으로 구분할 수 있다. 이는 전체 어플리케이션의 핵심 요구 사항과 기능들을 구분해서 모듈화할 수 있고 OOP에서라면 클래스와 컴포넌트 형태로 구성이 가능하다. 하지만 현실은 그렇지 못하다. 실제로 개발되어 돌아가는 각 모듈에는 해당 업무를 처리하기 위한 로직만 존재해서는 불완전할 수밖에 없다. 일단 각 업무를 처리하는 클래스와 구현된 메소드에는 향후 시스템을 분석하거나 추적을 위해 로그를 작성해야 하며, 인증받은 사용자가 접근하는지를 체크하고 권한 여부를 따지는 보안 기능이 필요하다. 또한 내부에서 사용하는 Persistence 처리를 위해 Transaction을 시작하고, 또 필요에 따라서 그것을 Commit 또는 Rollback하는 부분도 추가되어야 한다. 예외 상황이나 문제가 발생했을 때는 그것을 기록에 남기는 부분도 있어야 하고, 필요하면 관리자에게 이메일을 발송해야 한다.

이러한 부가적인 기능들은 독립적인 클래스로 구현될 수 있지만, 그렇게 구현된 기능들을 호출하고 사용하는 코드들이 핵심 모듈 안의 필요한 영역에 모두 포함될 수밖에 없다. 로깅, 인증, 권한체크, DB 연동, 트랜잭션, 락킹, 에러처리 등의 기능을 아무리 뛰어난 OOP 기술을 이용해 모듈로 구성하고 추상화를 통해 최대한 독립시킨다고 해도 핵심 모듈의 모든 클래스와 메소드 속에 이와 연동되는 부분이 매우 깊이 그리고 상당한 양을 갖으면서 자리 잡게 된다.



실제로 모듈화가 잘 된 어플리케이션 클래스를 보더라도 핵심 기능을 위한 코드보다 부가적인 기능과 처리를 위한 부분의 양이 더 많아지게 되는데 만약 다른 종류의 로깅 플랫폼을 사용해 로그 처리하는 클래스와 메소드가 달라지고 로그 메시지가 변경되어야 한다면 개발자들은 모든 클래스 안에 있는 로그 관련 코드를 일일이 다 수정해 주는 수밖에 없다. 그러다가 만약 중요한 클래스에서 한두 군데 로그 기록 코드가 빠졌고 이로 인해 결과를 확인하는데 문제가 생겼다면 이를 다시 확인하고 찾아내는 일만 해도 엄청난 작업이 아닐 수 없을 것이다. 이렇게 작성된 어플리케이션은 몇 가지 심각한 문제를 가지고 있다.

- **중복되는 코드** : 복사-붙이기에 의해 만들어진 여러 모듈에서 중복되는 코드의 문제점은 이미 잘 알려져 있다. 하지만 AOP를 사용하지 않은 대부분의 어플리케이션에서는 어떠한 추상화와 리팩토링을 통해서도 반복되는 코드를 피하기가 어렵다.
- **지저분한 코드** : Crosscutting Concerns과 관련된 코드들이 핵심 기능 코드 사이 사이에 끼어들어가 있기 때문에 코드가 지저분해지고 이에 따라 가독성이 떨어지며 개발자들의 실수나 버그를 유발하고 후에 코드를 유지보수하는데 큰 어려움을 준다.
- **생산성의 저하** : 어플리케이션 개발자들이 자주 등장하는 Crosscutting Concerns을 구현한 코드를 함께 작성해야 하기 때문에 개발의 집중력을 떨어뜨리고 결과적으로 전체 생산성의 저하를 가져온다. 또 모듈별로 개발자들을 구분하고 분산시키는 것에 한계가 있다.
- **재활용성의 저하** : OOP의 장점인 재활용성이 매우 떨어진다.
- **변화의 어려움** : 새로운 요구사항으로 인해 전체적으로 많은 부분에 영향을 미치는 경우 쉽게 새로운 요구사항을 적용하기 힘들게 된다. 또 새로운 관심 영역의 등장이나 이의 적용을 매우 어렵게 한다.

대표적인 AOP 툴

AOP는 OOP의 확장에 가깝기 때문에 전용 언어나 독립된 개발 툴을 가지고 있지 않고 대신 기존의 OOP를 확장한 언어 확장(languageextension) 또는 툴이나 프레임워크 형태로 사용할 수 있게 되어 있다. 대표적으로 AOP 구현의 시초가 된 Eclipse 프로젝트의 AspectJ를 들 수 있다. AspectJ는 초기에 제록스 PARC 연구소에서 개발되었다가 2002년에 이클립스 프로젝트에 기증되었고, 현재 IBM의 전폭적인 지원을 받으면서 개발되어 사용되고 있다. 그리고 BEA가 중심이 되어 개발하고 있는 AspectWerkz가 있다. AspectWerkz는 AspectJ와 달리 자바 언어 자체를 확장하지 않고 기존의 자바 언어만으로 AOP의 사용이 가능하도록 되어 있다. 그리고 의존성 삽입(Dependency Injection, 이하 DI) 기반의 프레임워크로 유명한 SpringAOP가 있다. 가장 최근에 등장한 AOP로는 JBossAOP도 있다. SpringAOP와 함께 대표적인 인터셉터체인 방식의 AOP로 꼽힌다.

	AspectJ	AspectWerkz	JBossAOP	SpringAOP
출시	2001	2002	2004	2004
버전	1.2.1	2.0	1.3.0	1.2.5
Aspect 선언	전용코드	XML, Annotation	XML, Annotation	XML
Advice	전용코드	자바 메소드	자바 메소드	자바 메소드
JoinPoint	메소드, 생성자, Advice, Field Access, 인스턴스	메소드, 생성자, Advice, Field Access, 인스턴스	메소드, 생성자, Advice, Field Access, 인스턴스	메소드
Pointcut 매칭	Signature, WildCard, Annotation	Signature, WildCard, Annotation	Signature, WildCard, Annotation	정규식
Weaving	컴파일 및 로딩 타임, 바이트 코드 생성	컴파일 및 로딩 타임, 바이트 코드 생성	런타임 인터셉션 및 Proxy	런타임 인터셉션 및 Proxy
IDE 지원	Eclipse, JDeveloper, JBuilder, NetBeans	Eclipse, NetBeans	Eclipse	

• AspectJ

AspectJ의 가장 큰 특징은 다른 AOP 툴과는 달리 자바 언어를 확장해서 만들어진 구조라는 것이다. 마치 새로운 AOP 언어를 사용하듯이 aspect라는 키워드를 이용해 Aspect나 Pointcut, Advice를 만들 수 있다. 따라서 일반 자바 컴파일러로는 컴파일이 불가능하고 특별한 AOP 컴파일러를 사용해야 한다. 하지만 이렇게 만들어진 바이너리는 표준 JVM에서 동작 가능한 구조로 되어있기 때문에 특별한 클래스 로더의 지원 없이도 실행 가능하다. AspectJ는 가장 오래되고 가장 많이 사용되는 AOP 툴이다. 동시에 가장 풍부한 기능을 가지고 있고 확장성이 뛰어나기 때문에 가장 이상적인 AOP 툴로 꼽히고 있다. 하지만 자바 언어를 확장했기 때문에 새로운 문법과 언어를 이해할 필요가 있고 프로젝트

트 빌드시 특별한 컴파일러를 사용해야 하는 불편함이 있다. Weaving이 컴파일시에 일어나기 때문에 Pointcut에 의해 선택된 모든 클래스들은 Aspect가 바뀔 때마다 모두 다시 컴파일되어야 한다.

• AspectWerkz

AspectWerkz는 AspectJ와는 달리 자바 언어를 확장하지 않는다. 따라서 표준 자바 클래스를 이용해서 AOP를 구현해 낼 수 있다. 일반 클래스와 메소드를 이용해 쉽게 구현이 가능한 Advice와 달리 복잡한 문법이 필요한 Pointcut은 별도의 XML 파일을 이용해 설정할 수 있도록 되어 있다. 자바 클래스와 XML 설정 파일의 접근법에 익숙한 개발자들에게는 매우 편리한 접근 방식이라고 볼 수 있다. 최근에는 JDK5의 지원에 따라 Annotation을 이용할 수 있어 더욱 편리해졌다. Weaving은 특별한 클래스 로더를 이용한 로딩타임 바이트코드 생성을 이용한다. AspectJ 못지않은 다양한 JoinPoint와 AOP 기능을 지원하고 있으며 편리한 개발을 위한 IDE 플러그인이 개발되어 있다.

• JBossAOP

JBossAOP는 기본적으로 컨테이너에서 동작하지만 컨테이너와 상관없는 독립된 자바 프로그램에서도 사용할 수 있다. 하지만 주 용도는 JBoss 서버와 앞으로 나올 EJB3 컨테이너 등에 AOP를 적용하는 데에 사용되어지는 것이다. AspectWerkz와 마찬가지로 Advice는 표준 자바 코드로 작성하고 Pointcut과 다른 설정은 XML 파일이나 JDK5의 Annotation으로 작성할 수 있다. 아직까지는 JBoss 사용자의 일부에서만 사용되고 있으나 향후 EJB3를 중심으로 한 POJO 기반의 엔터프라이즈 미들웨어 프레임워크가 개발되어짐에 따라 점차로 사용률이 올라갈 것으로 기대된다.

• SpringAOP

SpringAOP는 Spring Framework의 핵심기능 중의 한가지로 Spring의 Dependency Injection(이후 DI) 컨테이너에서 동작하는 엔터프라이즈 서비스에서 주로 사용된다. SpringAOP는 다른 AOP와 달리 기존 클래스의 바이트코드를 수정하지 않는다. 대신 JDK의 Dynamic Proxy나 CGLIB을 사용해서 Proxy 방식으로 AOP의 기능을 수행한다. 이 때문에 다른 AOP의 기능과 비교해서 매우 제한적인 부분만을 지원한다. 하지만 SpringAOP의 구현 목적은 엔터프라이즈 어플리케이션에서 주로 사용되는 핵심적인 기능에 AOP의 장점을 살려 이를 Spring 내에서 사용하는 것이기 때문에 다른 AOP와 같은 AOP의 복잡한 전체 기능을 굳이 다 필요로 하지 않는다. 프록시 기반의 SpringAOP는 SpringIoC/DI와 매우 긴밀하게 연동이 된다. 따라서 SpringAOP를 사용하는 방법은 Spring 내에 ProxyBean을 설정해서 쉽게 사용할 수 있다. JDK의 표준 기능만을 사용하기 때문에 특별한 빌드 과정이 필요없고 클래스 로더를 변경한다거나 하는 번거로운 작업이 없다. 대신 JoinPoint의 종류가 메소드 기반으로 제한되나 대부분의 엔터프라이즈 어플리케이션에서 필요로 하는 주요 AOP 기능들은 메소드 호출을 기반으로 충분히 처리가 가능하기 때문에 SpringAOP는 그 제한된 AOP 기능에도 불구하고 현장에서 가장 빠른 속도로 적용되어 사용되는 AOP 솔루션 중의 하나이다. SpringAOP는 Advice와 Pointcut을 모두 표준 자바 클래스로 작성할 수 있다. 필요에 따라서 Pointcut은 설정 파일 내에서 Pointcut FactoryBean을 이용해서 정규식으로 표현이 가능하다. SpringAOP의 최대 단점은 복잡한 Proxy 설정 구조이다. Spring Bean을 정의한 파일에서 Proxy를 정의한 부분의 다른 XML기반의 AOP에 비해서도 복잡한 편인데 이 경우 SpringAOP가 지원하는 AutoProxyingCreatorBean 등을 이용하면 설정 코드를 매우 단순하게 작성하는 것이 가능하다.



[참고] SpringAOP의 Proxy 기법

SpringAOP에서 제공하는 두 가지 Proxy 기법을 사용할 때 아래의 내용을 참고한다.

- 1. Proxy Target class가 Interface를 구현한 class인 경우 JDK Dynamic Proxy를 이용하여 객체를 Proxy 시킨다. Target class의 Interface가 존재하지 않는 경우, CGLIB을 이용하여 객체를 Proxy 시킨다.
- 2. proxyTargetClass 옵션을 true로 하는 경우 CGLIB 기반의 proxy 객체가 생성된다. proxyTargetClass 옵션이 false인 경우(default - false)라도 Target class의 Interface가 존재 하지 않는 경우에는 CGLIB 기반의 Proxy 객체가 생성된다.
- 3. CGLIB을 이용하여 Proxy 객체를 생성하는 경우 Default Constructor가 필요하다. (Spring 4.0 이상 버전에서는 Default Constructor가 없어도 Proxy 객체를 생성 할 수 있다.)

5.1.AOP 구성 요소

AOP에는 새로운 용어가 많이 등장한다. 이 중에서 특히 AOP를 이용해서 개발하는데 필요한 다음의 주요 구성 요소들에 대해 정확한 이해가 필요하다.

5.1.1.JointPoint

Crosscutting Concerns 모듈이 삽입되어 동작할 수 있는 실행 가능한 특정 위치를 말한다. 예를 들어 메소드가 호출되는 부분 또는 리턴되는 시점이 하나의 JoinPoint가 될 수 있다. 또 필드를 액세스하는 부분, 인스턴스가 만들어지는 지점, 예외가 던져지는 시점, 등이 대표적인 JoinPoint가 될 수 있다. 각각의 JoinPoint들은 그 전후로 Crosscutting Concerns의 기능이 AOP에 의해 자동으로 추가되어져서 동작할 수 있는 후보지가 되는 것이다.

5.1.2.Pointcut

Pointcut은 어느 JoinPoint를 사용할 것인지를 결정하는 선택 기능을 말한다. AOP가 항상 모든 모듈의 모든 JoinPoint를 사용할 것이 아니기 때문에 필요에 따라 사용해야 할 모듈의 특정 JoinPoint를 지정할 필요가 있다. 일종의 JoinPoint 선정 룰과 같은 개념으로 다음과 같은 Pattern Matching 방법들을 이용하여 룰을 정의할 수 있다.

5.1.2.1.Pattern Matching Examples

1. Basics

- **set*(..)** : set으로 시작하는 모든 메소드명
- *** main(..)** : return type이 any type이고, 0개 이상의 any type parameter를 가진 main 메소드

2. Matching Type

- **java.io.*** : java.io 패키지 내에 속한 모든 요소
- **org.myco.myapp.*** : org.myco.myapp 패키지 또는 서브 패키지 내에 속한 모든 요소
- **Number+** : Number 또는 Number의 서브 type으로 Integer, Float, Double ..등이 이에 해당
- **!(Number+)** : Number 또는 Number의 서브 type이 아닌 모든 type
- **org.xyz.myapp.* && !Serializable+** : org.xyz.myapp 패키지 또는 서브 패키지 내에 존재하면서 Serializable type이 아닌 모든 요소
- **int || Integer** : int 또는 Integer type

3. Matching Modifiers

- **public static void main(..)** : 0개 이상의 any type parameter를 가진 public static void main 메소드
- **!private * * (..)** : return type이 any type이고, 0개 이상의 any type parameter를 가진 모든 메소드 중 modifier가 private이 아닌 메소드
- *** main(..)** : modifier를 별도로 명시하지 않은 경우, default modifier가 아닌 any modifier 의미

4. Matching Parameter

- *** main(*)** : return type이 any type이고, 1개의 any type parameter를 가진 main 메소드
- *** main(*,..)** : return type이 any type이고, 최소 1개의 any type parameter를 가진 main 메소드

- *** main(*,...,String,*)** : return type이 any type이고, 최소 3개의 any type parameter를 가지며 끝에서 두번째 parameter type이 String인 main 메소드

5. Matching Constructor

- **new(..)** : 0개 이상의 any type parameter를 가진 constructor
- **Account.new(..)** : 0개 이상의 any type parameter를 가진 Account 클래스의 constructor

AspectJ는 Pointcut을 명시할 수 있는 다양한 Pointcut Designator(지시자)를 제공한다. 이제부터 앞서 정의한 Pattern Matching 방법을 이용하여, 본격적으로 Pointcut Designator별 Pointcut 정의 방법에 대해 살펴보기로 하자.

5.1.2.2. Pointcut Designators

1. execution 또는 call

특정 메소드나 생성자 실행을 위한 JoinPoint를 정의하는 것으로, JoinPoint의 특정 method name, parameter types, return type, declared exceptions, declaring type, modifiers에 대한 matching이 가능하며, 단, return type pattern, method name pattern, parameter list pattern은 필수적으로 정의해야 한다. 다음은 execution, call을 이용한 pointcut 정의 예이다.

- **execution(* main(..))** : return type이 any type이고, 0개 이상의 any type parameter를 가진 main 메소드 실행시
- **call(Account.new(..))** : any type parameter를 가진 Account 클래스의 constructor 호출시

2. get 또는 set

특정 Field에 접근하거나 특정 Field 수정을 위한 JoinPoint를 정의한다.

- **get(Collection + org.xyz.myapp..*.*)** : Collection type의 org.xyz.myapp 패키지에 속한 any field에 대한 getter 호출시
- **set(!private * Account+.*)** : Account type의 non-private field에 대한 setter 호출시

3. handler

Exception 핸들링을 위한 JoinPoint를 정의한다.

- **handler(DataAccessException)** : matches catch(DataAccessException){...} and doesn't match catch(RuntimeException)
- **handler(RuntimeException+)** : matches both

4. within

특정 유형에 속하는 JoinPoint를 정의하며, 주로&&, ||, ! 등과 함께 조합된 형태로 사용된다.

- **within(*)** : matches any JoinPoint
- **within(org.xyz.myapp..*)** : org.xyz.myapp 패키지 내에 속하는 모든 요소
- **within(Interface+)** : Interface type의 모든 요소

5. withincode

해당되는 메소드 또는 constructor 내에 정의된 코드를 위한 JoinPoint를 정의한다.

- **withincode(!void get*())** : return type이 void가 아니고 메소드명이 get으로 시작하며 parameter가 없는 메소드 내의 코드

6. args

입력값의 개수, type 등에 대한 JoinPoint를 정의한다.

- **call(* transfer(..) && args(DepositAccount,CheckingAccount,*))** : 메소드명이 transfer이고, 입력 인자가 2개 이상이며, 1,2번째 입력 인자의 type이 DepositAccount,CheckingAccount인 메소드 호출시

7. this

JoinPoint를 가진 object의 type을 정의한다. (Runtime type)

- **this(Account)** : 인터페이스 Account를 구현한 클래스(Proxy)의 모든 JoinPoint

8. target

JoinPoint를 가진 target object의 type을 정의한다. (Runtime type)

- **call(* *(..)) && target(Account)** : Account 클래스 내의 모든 메소드 호출시

Spring은 메소드 호출 부분에 대한 AOP만을 지원하므로, 위에 정의한 다양한 Pointcut Designator 중 execution, within, target, this, args만이 사용 가능하다.

5.1.3.Advice

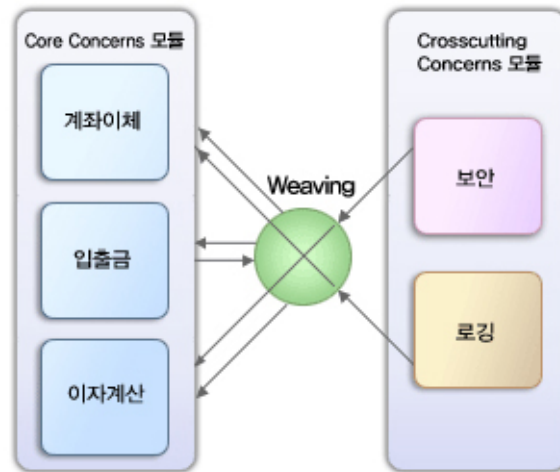
Advice는 각 JoinPoint에 삽입되어져 동작할 수 있는 코드로 동작 시점은 pointcut에 Matching되는 JoinPoint 실행 전후이며 **before, after, after returning, after throwing, around** 중에서 선택 가능하다.

Before	Before Advice는 Matching된 JoinPoint 전에 동작하는 Advice이다.
After	<p>After Advice는 동작 시점에 따라 after (finally), after returning, after throwing 으로 구분할 수 있다.</p> <ul style="list-style-type: none"> • after returning : Matching된 JoinPoint가 성공적으로 return된 후에 동작하는 Advice이다. • after throwing : Exception이 발생하여 Matching된 JoinPoint가 종료된 후에 동작하는 Advice이다. • after (finally) : Matching된 JoinPoint 종료 후에 동작하는 Advice이며 잘 사용되지 않는다.
Around	가장 강력한 Advice로 Matching된 JoinPoint 전, 후에 동작하며 JoinPoint 실행 시점을 결정할 수 있다. 또한 다른 Advice와는 달리 입력값, target object, return 값 등에 대한 변경이 가능하다.

동작 시점별 Advice 정의 방법에 대해서는 매뉴얼 >> Spring >> AOP 하위의 Annotation based AOP , XML based AOP , AspectJ based AOP 를 참고하도록 한다.

5.1.4.Weaving 또는 CrossCutting

AOP가 Core Concerns 모듈의 코드를 직접 건드리지 않고 필요한 기능이 작동하도록 하는 데는 Weaving 또는 CrossCutting이라고 불리는 특수한 작업이 필요하다. Core Concerns 모듈이 자신이 필요한 Crosscutting Concerns 모듈을 찾아 사용하는 대신에 AOP에서는 Weaving 작업을 통해 Core Concerns 모듈의 사이 사이에 필요한 Crosscutting Concerns 코드가 동작하도록 엮여지게 만든다. 이를 통해 AOP는 기존의 OOP로 작성된 코드들을 수정하지 않고도 필요한 Crosscutting Concerns 기능을 효과적으로 적용해 낼 수 있다.



Weaving은 기존의 자바 언어와 컴파일러에서는 쉽게 구현할 수 있는 방법이 아니었으며 본격적인 AOP 기술이 등장한 것은 1990년대 후반 제록스 PARC 연구소에서 그레거 키체일(Gregor Kiczales)에 의해 AspectJ가 개발되면서라고 볼 수 있다.

Weaving을 처리하는 방법은 다음과 같이 3가지가 존재한다.

Weaving 방식	설명
Compiletime Weaving	별도 컴파일러를 통해 Core Concerns 모듈의 사이 사이에 Aspect 형태로 만들어진 Crosscutting Concerns 코드들이 삽입되어 Aspect가 적용된 최종 바이너리가 만들어지는 방식이다. (ex. AspectJ, ...)
Loadingtime Weaving	별도의 Agent를 이용하여 JVM이 클래스를 로딩할 때 해당 클래스의 바이너리 정보를 변경한다. 즉, Agent가 Crosscutting Concerns 코드가 삽입된 바이너리 코드를 제공함으로써 AOP를 지원하게 된다. (ex. AspectWerkz, ...)
Runtime Weaving	소스 코드나 바이너리 파일의 변경없이 Proxy를 이용하여 AOP를 지원하는 방식이다. Proxy를 통해 Core Concerns를 구현한 객체에 접근하게 되는데, Proxy는 Core Concerns 실행 전후에 Cross Concerns를 실행한다. 따라서 Proxy 기반의 Runtime Weaving의 경우 메소드 호출시에만 AOP를 적용할 수 있다는 제한점이 있다. (ex. Spring AOP, ...)

5.1.5.Aspect

Aspect는 어디에서(Pointcut) 무엇을 할 것인지(Advice)를 합쳐놓은 것을 말한다. AspectJ와 같은 자바 언어를 확장한 AOP에서는 마치 자바의 클래스처럼 Aspect를 코드로 작성할 수 있다. 다음은 모든 클래스의 main 메소드 실행(pointcut main()) 후에 "Hello from AspectJ"라는 문자열을 남기는 (after returning advice) Aspect HelloFromAspectJ의 일부이다.

```
public aspect HelloFromAspectJ{
    // define pointcut
    pointcut main(): execution(public static void main(String[]));
    // define advice
    after() returning : main() {
        System.out.println("Hello from AspectJ!");
    }
}
```

Aspect 정의에 대한 자세한 설명은 매뉴얼 >> Spring >> AOP 하위의 Annotation based AOP , XML based AOP , AspectJ based AOP 를 참고하도록 한다.

5.2.Annotation based AOP

다음에서는 AOP 대표적인 툴 중 @AspectJ(Annotation)을 이용하여 Aspect를 정의하고 테스트하는 방법에 대해서 다루고자 한다. @AspectJ(Annotation)은 AspectJ 5 버전에 추가된 Annotation이며, Spring 2.0에서부터 이러한 Annotation에 대한 처리가 가능하므로, Spring 기반일 경우 별도의 Compiler나 Weaver 없이 @AspectJ(Annotation) 기반의 AOP 적용이 가능하다. 또한 Annotation을 이용하여 Aspect를 정의할 경우 별도 XML 파일에 대한 정의가 불필요하므로, Aspect 적용이 보다 간결해짐을 알 수 있을 것이다. (단, Annotation은 JAVA 5 이상에서만 정의 가능함에 유의하도록 한다.)

5.2.1.Configuration

@AspectJ(Annotation)이 적용된 클래스들을 로딩하여 해당 클래스에 정의된 Pointcut, Advice를 실행하기 위해서는 Spring 속성 정의 XML 파일에 다음과 같이 추가해주어야 한다.

```
<aop:aspectj-autoproxy/>
```

5.2.2.@Aspect 정의

@Aspect를 이용하여 특정 클래스가 Aspect임을 나타낸다. 다음 LoggingAspect에서는 @Aspect를 이용하여 해당 클래스가 Aspect임을 나타내고 있다.

```
@Aspect
public class LoggingAspect {
    //...
}
```

5.2.3.@Pointcut 정의

@Pointcut을 이용하여 해당 Aspect를 적용할 부분을 정의한다. (Pointcut 정의시에는 Pointcut Designator와 Pattern Matching 활용 방법을 참고한다.) 다음은 PrintStringUsingAnnotation의 Pointcut 정의 부분이다. @Pointcut을 "execution(* org.anyframe.sample..*Impl.*(..))"와 같이 정의하고, 해당 Pointcut에 대해 식별자로서 serviceMethod라는 메소드명을 부여하였다. 이것은 클래스명이 Impl로 끝나는 모든 메소드의 실행 부분이 Aspect를 적용할 Pointcut임을 의미한다. 해당 Pointcut은 serviceMethod()라는 이름으로 이용 가능하다.

```
@Pointcut("execution(* org.anyframe.sample..*Impl.*(..))")
public void serviceMethod(){}

@Before("serviceMethod()")>
public void beforeLogging(JoinPoint thisJoinPoint) {
    ...
}
```

또는 @Pointcut을 생략하고 @Before, @AfterReturning, @AfterThrowing, @After의 속성으로 바로 Pointcut을 정의할 수 있다. 축약된 표현의 장점은 별도의 메소드 정의(serviceMethod())와 같은를 하지 않아도 된다는 것이며 이에 대한 예제는 다음절에서 자세히 살펴보도록 하겠다.

5.2.4.@Advice 정의

다음에서는 Annotation을 이용하여 동작 시점별 Advice를 정의하는 방법에 대해 살펴보기로 한다.

5.2.4.1.Before Advice

@Before를 이용하여 Before Advice를 정의한다. 다음은 Before Advice 정의 부분이다. Before Advice인 beforeLogging()는 앞서 정의한 serviceMethod()라는 Pointcut 전에 "Logging Aspect : executed "라는 문자와 해당 Pointcut을 가진 메소드명 클래스명을 출력하는 역할을 수행한다.

```
@Before("serviceMethod()")
public void beforeLogging(JoinPoint thisJoinPoint) {
    Class<? extends Object> clazz = thisJoinPoint.getTarget().getClass();
    String methodName = thisJoinPoint.getSignature().getName();
    Object[] arguments = thisJoinPoint.getArgs();

    StringBuilder argBuf = new StringBuilder();
    StringBuilder argValueBuf = new StringBuilder();
    int i = 0;
    for (Object argument : arguments) {
        String argClassName = argument.getClass().getSimpleName();
        if (i > 0) {
            argBuf.append(", ");
        }
        argBuf.append(argClassName + " arg" + ++i);
        argValueBuf.append(".arg" + i + " : " + argument.toString() + "\n");
    }

    if (i == 0) {
        argValueBuf.append("No arguments\n");
    }

    StringBuilder messageBuf = new StringBuilder();
    messageBuf.append("before executing {} ({{}) method");

    messageBuf.append("\n-----\n");
    messageBuf.append(" {{}} ");
    messageBuf.append("-----");

    Logger logger = LoggerFactory.getLogger(clazz);
    logger.debug(
        messageBuf.toString(),
        new Object[]{methodName, argBuf.toString(), argValueBuf.toString()});
}
```

beforeLogging()는 1개의 입력 인자(JoinPoint)를 가지고 있는데 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있다. Target 정보가 불필요한 Advice인 경우에는 JoinPoint라는 입력 인자를 선언하지 않아도 된다.

또는 @Pointcut 태그를 생략하고 다음과 같이 @Before 태그내의 속성을 이용하여 축약형으로 정의할 수도 있다. 이 경우 serviceMethod()와 같이 @Pointcut 태그를 지정하기 위해 사용했던 메소드 정의는 생략된다.

```
@Before("execution(* org.anyframe.sample.*Impl.*(..))")
public void beforeLogging(JoinPoint thisJoinPoint) {
    ...
}
```

5.2.4.2.AfterReturning Advice

@AfterReturning을 이용하여 AfterReturning Advice를 정의한다. 다음은 AfterReturning Advice 정의 부분으로 해당 Pointcut 실행 결과를 retVal이라는 변수에 담도록 정의하고 있다. AfterReturning

Advice인 `afterReturningExecuteGetMethod()`는 앞서 정의한 Pointcut 후에, "AfterReturning Advice of `PrintStringUsingAnnotation`"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
@AfterReturning(pointcut = "execution(* org.anyframe.sample..*Impl.*(..))", returning =
    "retVal")
public void afterReturningExecuteGetMethod(JoinPoint thisJoinPoint, Object retVal) {
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("AfterReturning Advice of PrintStringUsingAnnotation");
    System.out.println("***" + targetClass + "." + opName + "()" + "***");
}
```

`afterReturningExecuteGetMethod()`는 2개의 입력 인자(`JoinPoint`, `Object`)를 가지고 있는데 첫번째 인자는 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있으며, 두번째 인자는 해당 Pointcut의 실행 결과이다. AfterReturning Advice에서 특정 Pointcut 실행 결과를 참조해야 한다면, Advice 정의시 `returning`의 값을 정의하고 해당하는 메소드의 입력 인자명을 동일하게 정의해주도록 한다. 각 입력 인자는 AfterReturning Advice 정의시 필요에 따라 선택 정의할 수 있다.

5.2.4.3.AfterThrowing Advice

@AfterThrowing을 이용하여 AfterThrowing Advice를 정의한다. 다음은 `transfer`의 AfterThrowing Advice 정의 부분으로 해당 Pointcut 실행시 발생한 Exception 객체를 `exception`이라는 변수에 담도록 정의하고 있다. AfterThrowing Advice는 정의한 Pointcut에서 Exception이 발생한 후에 Exception을 핸들링하고 Exception의 종류에 따라 Exception message를 출력하게 된다.

```
@AfterThrowing(pointcut = "execution(* org.anyframe.sample..*Impl.*(..))", throwing =
    "exception")
public void transfer(JoinPoint thisJoinPoint, Exception exception) throws
    MovieFinderException {
    Object target = thisJoinPoint.getTarget();
    while (target instanceof Advised) {
        try {
            target = ((Advised) target).getTargetSource().getTarget();
        } catch (Exception e) {
            LogFactory.getLogger(this.getClass()).error("Fail to get target object
                from JointPoint.", e);
            break;
        }
    }

    String className = target.getClass().getSimpleName().toLowerCase();
    String opName = (thisJoinPoint.getSignature().getName()).toLowerCase();
    Log logger = LogFactory.getLogger(target.getClass());

    if (exception instanceof MovieFinderException) {
        MovieFinderException empEx = (MovieFinderException) exception;
        logger.error(empEx.getMessage(), empEx);
        throw empEx;
    }

    try {
        logger.error(messageSource.getMessage("error." + className + "."
            + opName, new String[] {}, Locale.getDefault()), exception);
    } catch (Exception e) {
        logger.error(messageSource.getMessage("error.common",
            new String[] {}, Locale.getDefault()), exception);
        throw new MovieFinderException(messageSource, "error.common");
    }
}
```

```

    }
    throw new MovieFinderException(messageSource, "error." + className
        + "." + opName);
}

```

transfer()는 2개의 입력 인자(JoinPoint, Exception)를 가지고 있는데 첫번째 인자는 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있으며, 두번째 인자는 Pointcut 실행시 발생한 Exception 객체이다. AfterThrowing Advice에서 특정 Pointcut 실행시 발생한 Exception을 참조해야 한다면, Advice 정의시 throwing의 값을 정의하고 해당하는 메소드의 입력 인자명을 동일하게 정의해주도록 한다. 각 입력 인자는 AfterThrowing Advice 정의시 필요에 따라 선택 정의할 수 있다.

5.2.4.4.After(finally) Advice

@After를 이용하여 After(finally) Advice를 정의한다. 다음은 PrintStringUsingAnnotation 의 After(finally) Advice 정의 부분이다. After(finally) Advice인 afterExecuteGetMethod()는 get 메소드들을 정의한 Pointcut 후에 "After(finally) Advice of PrintStringUsingAnnotation"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```

@After("execution(* org.anyframe.sample.*Impl.get*(..))")
public void afterExecuteGetMethod(JoinPoint thisJoinPoint) {
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("After(finally) Advice of PrintStringUsingAnnotation");
    System.out.println("****" + targetClass + "." + opName + "()" + "****");
}

```

afterExecuteGetMethod()는 1개의 입력 인자(JoinPoint)를 가지고 있는데 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있다. Target 정보가 불필요한 Advice인 경우에는 JoinPoint라는 입력 인자를 선언하지 않아도 된다.

5.2.4.5.Around Advice

@Around를 이용하여 Around Advice를 정의한다. 다음은 PrintStringAroundUsingAnnotation 의 Around Advice 정의 부분이다. Around Advice인 aroundExecuteUpdateMethod()는 update 메소드를 정의한 Pointcut 후에 "Around Advice of PrintStringUsingAnnotation"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```

@Around("execution(* org.anyframe.sample.*Impl.update*(..))")
public Object aroundExecuteUpdateMethod(ProceedingJoinPoint thisJoinPoint) throws Throwable
{
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("Around Advice of PrintStringUsingAnnotation");
    System.out.println("****" + targetClass + "." + opName + "()" + "****");

    // before logic
    Object retVal = thisJoinPoint.proceed();
    // after logic

    return retVal;
}

```

aroundExecuteUpdateMethod()는 1개의 입력 인자(ProceedingJoinPoint)를 가지고 있는데 proceed()라는 메소드 호출을 통해 대상 Pointcut을 실행할 수 있으며, Target 클래스명, 메소드명 등과 같은 Target

정보도 포함하고 있다. 즉, Pointcut 전, 후 처리가 가능하며, Pointcut 실행 시점을 결정할 수 있다. 또한 다른 Advice와는 달리 입력값, target, return 값 등에 대해 변경이 가능하다. Target 정보가 불필요한 Advice인 경우에는 ProceedingJoinPoint라는 입력 인자를 선언하지 않아도 된다.

5.2.5.Aspect 실행

이제 테스트코드 Main.java를 이용하여 앞서 언급한 Aspect들이 정상적으로 동작하는지 확인해 보도록 하자. 다음은 테스트코드 Main.java 의 main()메소드로 실제 movie에 대한 CRUD 로직을 수행함으로써 Befor Advice로 정의된 LoggingAspect가 제대로 수행되는지 확인할 수 있다.

```
public static void main(String[] args) throws Exception {
    Main main = new Main();
    // 1. initialize context
    main.setup();
    // 2. test
    main.manageMovie();
    // 3. close context
    main.teardown();
}

public void manageMovie() throws Exception {
    // 1. lookup aopMovieFinder, aopMovieService
    MovieFinder movieFinder = (MovieFinder) context.getBean("aopMovieFinder");
    MovieService movieService = (MovieService) context.getBean("aopMovieService");

    // 2. create a new movie
    Movie movie = getMovie();
    movieService.create(movie);

    // 3. get movie list
    Page movies = movieFinder.getPagingList(movie, 1);

    // 4. update movie
    movie.setTitle("Mission Impossible");
    movieService.update(movie);

    // 5. get movie
    Movie result = movieService.get(movie.getMovieId());

    // 6. remove movie
    movieService.remove(movie.getMovieId());
}

private Movie getMovie() throws Exception {
    Genre genre = new Genre();
    genre.setGenreId("GR-03");

    Movie movie = new Movie();
    movie.setMovieId("MV-99999");
    movie.setTitle("Shrek (2001)");
    movie.setActors("Shrek");
    movie.setDirector("Andrew Adamson");
    movie.setGenre(genre);
    movie.setReleaseDate(new Date());
    movie.setRuntime(new Long(90));
    movie.setTicketPrice(new Float(8000));
    movie.setNowPlaying("N");

    return movie;
}
```


첫번째 로직 movieService.create(movie); 실행시 Before Advice로 정의된 LoggingAspect 클래스가 적용되며, 콘솔창에 다음과 같은 실행 결과를 포함하게 된다.

```
before executing create(Movie arg1) method
-----
.arg1 : [actors=Shrek, director=Andrew Adamson, movieId=MV-99999, nowPlaying=N,
posterFile=null, releaseDate=Mon Apr 18 13:40:42 KST 2011, runtime=90, ticketPrice=8000.0,
title=Shrek (2001), genreId=GR-03]
-----
```

두번째 로직 movieFinder.getPagingList(movie, 1); 실행시 Before Advice로 정의된 LoggingAspect 클래스가 적용되며, 콘솔창에 다음과 같은 실행 결과를 포함하게 된다.

```
before executing getPagingList(Movie arg1, Integer arg2) method
-----
.arg1 : [actors=Shrek, director=Andrew Adamson, movieId=MV-99999, nowPlaying=N,
posterFile=null, releaseDate=Mon Apr 18 13:40:42 KST 2011, runtime=90, ticketPrice=8000.0,
title=Shrek (2001), genreId=GR-03]
.arg2 : 1
-----
```

5.3.XML based AOP

다음에서는 AOP 대표적인 툴 중 Spring AOP를 이용하여 XML 스키마 기반에서 Aspect를 정의하고 테스트하는 방법에 대해서 다루고자 한다. Spring 2.x 버전부터 AOP 설정을 위한 aop namespace, XML 스키마가 추가되었다.

5.3.1.Aspect 정의

<aop:config>의 하위 태그인 <aop:aspect>를 이용하여 Aspect을 정의한다. 다음 context-aspect.xml 에서는 aop namespace를 이용하여 Aspect을 정의하고 있다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

    <bean id="methodLoggingAspect"
class="org.anyframe.sample.aop.common.aspect.LoggingAspect" />
    <bean id="exceptionTransfer"
class="org.anyframe.sample.aop.common.aspect.ExceptionTransfer">
        <property name="messageSource" ref="messageSource"/>
    </bean>
</beans>
```

Advice를 정의한 클래스를 Bean으로 정의해 두고, 해당 Bean을 <aop:config>내의 <aop:aspect> 에서 참조하는 형태로 Aspect을 정의할 수 있다.

5.3.2.Pointcut 정의

<aop:pointcut> 내의 expression의 값에 Pointcut Designator와 Pattern Matching을 이용하여 pointcut을 정의한다. 그리고 id의 값에 식별자를 부여한다. (Pointcut 정의시에는 Pointcut Designator와 Pattern Matching 활용 방법을 참고한다.)

```
<aop:pointcut id="serviceMethod" expression="execution(*
org.anyframe.sample..*Impl.*(..))" />
```

이것은 클래스명이 Impl로 끝나는 모든 메소드의 실행 부분이 Aspect를 적용할 Pointcut임을 의미한다. 해당 Pointcut은 serviceMethod라는 이름으로 이용 가능하다.

5.3.3.Advice 정의 및 구현

다음에서는 XML 기반에서 동작 시점별로 Advice 정의 및 구현 방법에 대해 살펴보기로 한다.

5.3.3.1.Before Advice

<aop:before>를 이용하여 Before Advice를 정의한다. 다음은 context-aspect.xml 의 Before Advice 정의 부분이다. 앞서 정의한 serviceMethod pointcut을 참조하고 있으며, 해당 pointcut 전에 methodLoggingAspect라는 Bean의 beforeLogging() 메소드를 호출해야 함을 명시하고 있다.

```
<aop:before method="beforeLogging" pointcut-ref="serviceMethod"/>
```

다음은 Before Advice를 구현하고 있는 LoggingAspect 클래스의 일부이다. Before Advice 역할을 수행하는 beforeLogging()는 앞서 정의한 serviceMethod Pointcut 전에 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
public class LoggingAspect {
    public void beforeLogging(JoinPoint thisJoinPoint) {
        Class<? extends Object> clazz = thisJoinPoint.getTarget().getClass();
        String methodName = thisJoinPoint.getSignature().getName();
        Object[] arguments = thisJoinPoint.getArgs();

        StringBuilder argBuf = new StringBuilder();
        StringBuilder argValueBuf = new StringBuilder();
        int i = 0;
        for (Object argument : arguments) {
            String argClassName = argument.getClass().getSimpleName();
            if (i > 0) {
                argBuf.append(", ");
            }
            argBuf.append(argClassName + " arg" + ++i);
            argValueBuf.append(".arg" + i + " : " + argument.toString() + "\n");
        }

        if (i == 0) {
            argValueBuf.append("No arguments\n");
        }

        StringBuilder messageBuf = new StringBuilder();
        messageBuf.append("before executing {} ({{}} method");

        messageBuf.append("\n-----\n");
        messageBuf.append(" {}");
        messageBuf.append("-----");

        Logger logger = LoggerFactory.getLogger(clazz);
        logger.debug(
            messageBuf.toString(),
            new Object[]{methodName, argBuf.toString(), argValueBuf.toString()});
    }
}
```

beforeLogging()는 1개의 입력 인자(JoinPoint)를 가지고 있는데 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있다. Target 정보가 불필요한 Advice인 경우에는 JoinPoint라는 입력 인자를 선언하지 않아도 된다.

5.3.3.2.AfterReturning Advice

<aop:after-returning>을 이용하여 AfterReturning Advice를 정의한다. 다음은 context-aspect.xml 의 AfterReturning Advice 정의 부분이다. 앞서 정의한 serviceMethod라는 pointcut을 참조하고 있으며, 해당 pointcut 후에 printStringAspect라는 Bean의 afterReturningExecuteGetMethod() 메소드를 호출해야 함을 명시하고 있다. 또한 해당 Pointcut 실행 결과를 retVal이라는 변수에 담도록 하고 있다.

```
<aop:after-returning method="afterReturningExecuteGetMethod" returning="retVal"
    pointcut-ref="serviceMethod" />
```

다음은 AfterReturning Advice를 구현하고 있는 PrintStringUsingXML 클래스의 일부이다. AfterReturning Advice 역할을 수행하는 afterReturningExecuteGetMethod()는 앞서 정의한 Pointcut 후에, "AfterReturning Advice of PrintStringUsingXML"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
public class PrintStringUsingXML {
    // ...

    public void afterReturningExecuteGetMethod(JoinPoint thisJoinPoint, Object retVal) {
        Class targetClass = thisJoinPoint.getTarget().getClass();
        Signature signature = thisJoinPoint.getSignature();
        String opName = signature.getName();

        System.out.println("AfterReturning Advice of PrintStringUsingXML");
        System.out.println("****" + targetClass + "." + opName + "()" + "****");
    }

    // ...
}
```

afterReturningExecuteGetMethod()는 2개의 입력 인자(JoinPoint, Object)를 가지고 있는데 첫번째 인자는 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있으며, 두번째 인자는 해당 Pointcut의 실행 결과이다. AfterReturning Advice에서 특정 Pointcut 실행 결과를 참조해야 한다면, XML에 해당 Advice 정의시 returning의 값을 정의하고 해당하는 메소드의 입력 인자명을 동일하게 정의해주도록 한다. 각 입력 인자는 AfterReturning Advice 정의시 필요에 따라 선택 정의할 수 있다.

5.3.3.3.AfterThrowing Advice

<aop:throwing>을 이용하여 AfterThrowing Advice를 정의한다. 다음은 context-aspect.xml 의 AfterThrowing Advice 정의 부분이다. 앞서 정의한 serviceMethod라는 pointcut을 참조하고 있으며, 해당 pointcut 후에 exceptionTransfer Bean의 tranfer() 메소드를 호출해야 함을 명시하고 있다. 또한 해당 Pointcut 실행시 발생한 Exception을 exception이라는 변수로 해당 Advice의 입력 인자명과 동일해야 한다.

```
<aop:after-throwing throwing="exception" pointcut-ref="serviceMethod" method="transfer" />
```

다음은 AfterThrowing Advice를 구현하고 있는 ExceptionTransfer 클래스의 일부이다. AfterThrowing Advice 역할을 수행하는 transfer()는 앞서 정의한 Pointcut에서 Exception이 발생한 후에 에러메시지를 출력하는 역할을 수행한다.

```
public class ExceptionTransfer {

    private MessageSource messageSource;
```

```

public void setMessageSource(MessageSource messageSource) {
    this.messageSource = messageSource;
}

public void transfer(JoinPoint thisJoinPoint, Exception exception) throws
MovieFinderException {
    Object target = thisJoinPoint.getTarget();
    while (target instanceof Advised) {
        try {
            target = ((Advised) target).getTargetSource().getTarget();
        } catch (Exception e) {
            LogFactory.getLogger(this.getClass()).error(
                "Fail to get target object from JointPoint.", e);
            break;
        }
    }

    String className = target.getClass().getSimpleName().toLowerCase();
    String opName = (thisJoinPoint.getSignature().getName()).toLowerCase();
    Log logger = LogFactory.getLogger(target.getClass());

    if (exception instanceof MovieFinderException) {
        MovieFinderException empEx = (MovieFinderException) exception;
        logger.error(empEx.getMessage(), empEx);
        throw empEx;
    }

    try {
        logger.error(messageSource.getMessage("error." + className + "."
            + opName, new String[] {}, Locale.getDefault()), exception);
    } catch (Exception e) {
        logger.error(messageSource.getMessage("error.common",
            new String[] {}, Locale.getDefault()), exception);
        throw new MovieFinderException(messageSource, "error.common");
    }
    throw new MovieFinderException(messageSource, "error." + className
        + "." + opName);
}
}

```

transfer()는 2개의 입력 인자(JoinPoint, Exception)를 가지고 있는데 첫번째 인자는 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있으며, 두번째 인자는 Pointcut 실행시 발생한 Exception 객체이다. AfterThrowing Advice에서 특정 Pointcut 실행시 발생한 Exception을 참조해야 한다면, XML에 해당 Advice 정의시 throwing의 값을 정의하고 해당하는 메소드의 입력 인자명을 동일하게 정의해주도록 한다. 각 입력 인자는 AfterThrowing Advice 정의시 필요에 따라 선택 정의할 수 있다.

5.3.3.4.After(finally) Advice

<aop:after>를 이용하여 After(finally) Advice를 정의한다. 다음은 context-aspect.xml 의 After(finally) Advice 정의 부분이다. 앞서 정의한 getMethods라는 pointcut을 참조하고 있으며, 해당 pointcut 후에 printStringAspect라는 Bean의 afterExecuteGetMethod() 메소드를 호출해야 함을 명시하고 있다.

```
<aop:after method="afterExecuteGetMethod" pointcut-ref="getMethods">
```

다음은 After(finally) Advice를 구현하고 있는 PrintStringUsingXML 클래스의 일부이다. After(finally) Advice 역할을 수행하는 afterExecuteGetMethod()는 앞서 정의한 getMethods()라는 Pointcut 후에 "After(finally) Advice of PrintStringUsingXML"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
public class PrintStringUsingXML {
```

```
// ...

public void afterExecuteGetMethod(JoinPoint thisJoinPoint) {
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("After(finally) Advice of PrintStringUsingXML");
    System.out.println("***" + targetClass + "." + opName + "()" + "***");
}

// ...
}
```

afterExecuteGetMethod()는 1개의 입력 인자(JoinPoint)를 가지고 있는데 Target 클래스명, 메소드명 등과 같은 Target 정보를 포함하고 있다. Target 정보가 불필요한 Advice인 경우에는 JoinPoint라는 입력 인자를 선언하지 않아도 된다.

5.3.3.5.Around Advice

<aop:around>를 이용하여 Around Advice를 정의한다. 다음은 context-aspect.xml 의 Around Advice 정의 부분이다. updateMethods라는 pointcut을 참조하고 있으며, 해당 pointcut 후에 printStringAspect라는 Bean의 aroundExecuteGetMethod() 메소드를 호출해야 함을 명시하고 있다.

```
<aop:around method="aroundExecuteUpdateMethod" pointcut-ref="updateMethods">
```

다음은 Around Advice를 구현하고 있는 PrintStringUsingXML 클래스의 일부이다. Around Advice 역할을 수행하는 aroundExecuteUpdateMethod()는 updateMethods()라는 Pointcut 후에 "Around Advice of PrintStringUsingXML"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
public class PrintStringUsingXML {
    // ...

    public Object aroundExecuteUpdateMethod(ProceedingJoinPoint thisJoinPoint)
        throws Throwable {
        Class targetClass = thisJoinPoint.getTarget().getClass();
        Signature signature = thisJoinPoint.getSignature();
        String opName = signature.getName();

        System.out.println("Around Advice of PrintStringUsingXML");
        System.out.println("***" + targetClass + "." + opName + "()" + "***");
        // before logic
        Object retVal = thisJoinPoint.proceed();
        // after logic
        return retVal;
    }

    // ...
}
```

aroundExecuteUpdateMethod()는 1개의 입력 인자(ProceedingJoinPoint)를 가지고 있는데 proceed()라는 메소드 호출을 통해 대상 Pointcut을 실행할 수 있으며, Target 클래스명, 메소드명 등과 같은 Target 정보도 포함하고 있다. 즉, Pointcut 전, 후 처리가 가능하며, Pointcut 실행 시점을 결정할 수 있다. 또한 다른 Advice와는 달리 입력값, target, return 값 등에 대해 변경이 가능하다. Target 정보가 불필요한 Advice인 경우에는 ProceedingJoinPoint라는 입력 인자를 선언하지 않아도 된다.

5.4.AspectJ based AOP

다음에서는 AOP 대표적인 툴 중 AspectJ를 이용하여 Aspect를 정의하고 테스트하는 방법에 대해서 다루고자 한다.

5.4.1.시작하기 전에

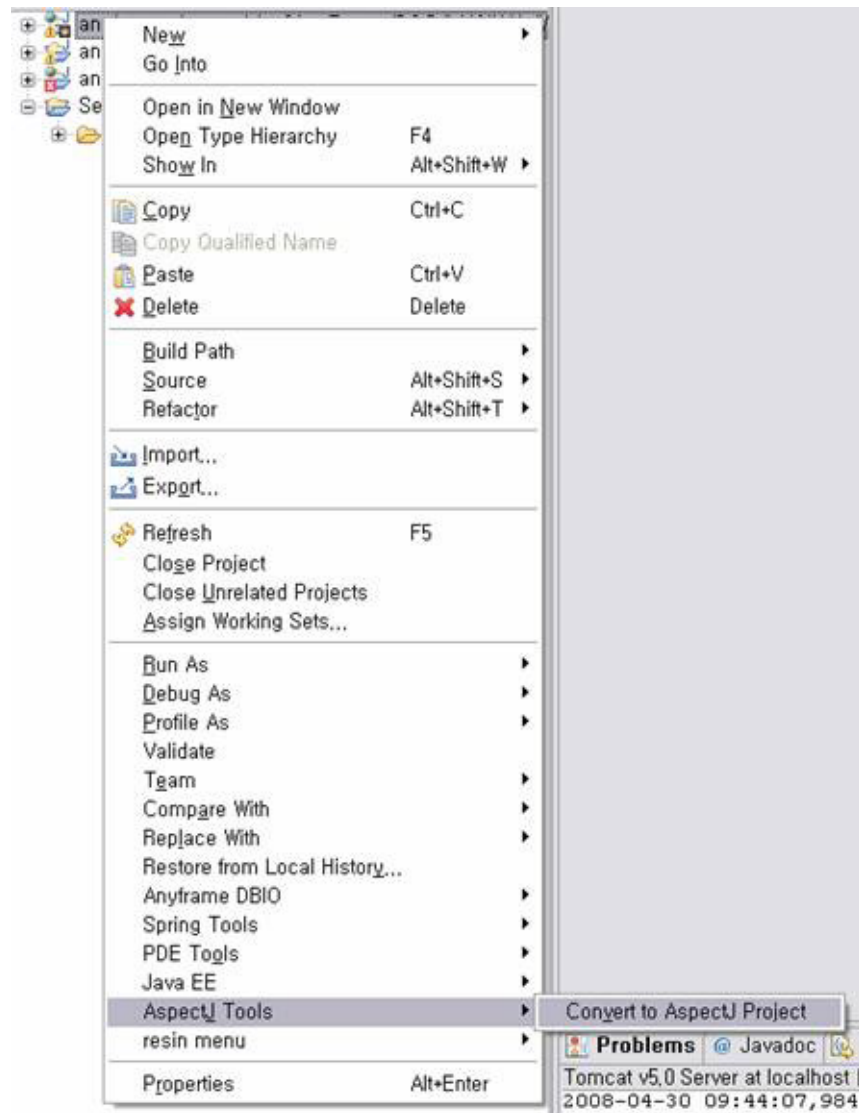
AspectJ를 이용하기 위해서는 다음과 같은 사항에 대해 확인이 필요하다.

1. AJDT(AspectJ Development Tool) 설치 Eclipse 플러그인 AJDT는 Aspect 파일을 생성하고 컴파일하기 위한 개발툴이다. 사용중인 Eclipse 내에 플러그인 AJDT(AspectJ Development Tool)가 설치되어 있지 않다면, AJDT(AspectJ Development Tool)를 다운로드 하여 사용중인 Eclipse 내에 설치하는 것이 좋다. 만약, AJDT를 이용하지 않고 Aspect를 컴파일하고자 한다면, aspectjtools-1.5.4.jar 내에 정의된 Ant task "iajc"을 이용하도록 한다. 다음은 샘플 build.xml 파일의 compile target의 내용이다.

```
<target name="compile" depends="init">
  <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
    <classpath>
      <pathelement location="${lib.dir}/aspectjtools-1.5.4.jar" />
    </classpath>
  </taskdef>

  <iajc verbose="true" destdir="${output.dir}" debug="on" source="1.5"
        showweaveinfo="true" xnoinline="true">
    <sourceroots>
      <pathelement location="src/main/java" />
    </sourceroots>
    <classpath>
      <pathelement location="${lib.dir}/aspectjrt-1.5.4.jar" />
      <pathelement location="${lib.dir}/slf4j-api-1.6.4.jar" />
      <pathelement location="${lib.dir}/slf4j-log4j12-1.6.4.jar" />
    </classpath>
  </iajc>
</target>
```

2. Convert to AspectJ Project 특정 프로젝트가 확장자 aj를 가진 Aspect 파일을 인식할 수 있도록 하기 위해서는 해당 프로젝트에 대한 context menu AspectJ Tools > Convert to AspectJ Project를 선택하여 해당 프로젝트에 대해 AspectJ 프로젝트의 성격을 부여해 주어야 한다.



5.4.2.Aspect 정의

확장자가 aj인 파일을 생성하고, aspect을 이용하여 Aspect 클래스를 정의한다. 다음 LoggingAspect에서는 aspect를 이용하여 해당 클래스가 Aspect임을 나타내고 있다.

```
public aspect LoggingAspect {
    // ...
}
```

5.4.3.Pointcut 정의

pointcut을 이용하여 해당 Aspect를 적용할 부분을 정의한다. (Pointcut 정의시에는 Pointcut Designator와 Pattern Matching 활용 방법을 참고한다.) 다음은 LoggingAspect의 Pointcut 정의 부분이다. pointcut을 "execution(execution(* org.anyframe.sample..*Impl.*(..)))"와 같이 정의하고, 해당 Pointcut에 대해 식별자로서 serviceMethod()라는 메소드명을 부여하였다. 이것은 클래스명이 GenericService로 끝나는 모든 메소드의 실행 부분이 Aspect를 적용할 Pointcut임을 의미한다. 해당 Pointcut은 serviceMethod()라는 이름으로 이용 가능하다.

```
pointcut serviceMethod(): execution(* org.anyframe.sample..*Impl.*(..));
```

5.4.4.Advice 정의

다음에서는 AspectJ 기반에서 동작 시점별 Advice를 정의하는 방법에 대해 살펴보기로 한다.

5.4.4.1.Before Advice

before()를 이용하여 Before Advice를 정의한다. 다음은 PrintStringUsingAspectJ 의 Before Advice 정의 부분이다. Before Advice는 앞서 정의한 getMethods()라는 Pointcut 전에 "Before Advice of PrintStringUsingAspectJ"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
before() : getMethods(){
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("Before Advice of PrintStringUsingAspectJ");
    System.out.println("****" + targetClass + "." + opName + "()" + "****");
}
```

위에서 제시한 Before Advice는 내부에 정의된 JoinPoint 유형의 thisJoinPoint라는 객체를 이용하여, Target 클래스명, 메소드명 등과 같은 Target 정보를 추출하고 있다.

5.4.4.2.AfterReturning Advice

after() returning()을 이용하여 AfterReturning Advice를 정의한다. 다음은 PrintStringUsingAspectJ 의 AfterReturning Advice 정의 부분으로 해당 Pointcut 실행 결과를 retVal이라는 변수에 담도록 정의하고 있다. AfterReturning Advice는 앞서 정의한 Pointcut 후에 , "AfterReturning Advice of PrintStringUsingAspectJ"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
after() returning(UserVO retVal) : getMethods()
{
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("AfterReturning Advice of PrintStringUsingAspectJ");
    System.out.println("****" + targetClass + "." + opName + "()" + "****");
}
```

위에서 제시한 AfterReturning Advice는 내부 정의된 JoinPoint 유형의 thisJoinPoint라는 객체를 이용하여, Target 클래스명, 메소드명 등과 같은 Target 정보를 추출하고 있다. 또한, 1개의 입력 인자(UserVO)를 가지고 있는데 이는 해당 Pointcut의 실행 결과이다. AfterReturning Advice에서 특정 Pointcut 실행 결과를 참조해야 한다면, Advice 정의시 returning에 해당하는 객체를 정의하고 메소드 로직 내에서 이를 활용하면 된다. 입력 인자는 AfterReturning Advice 정의시 필요에 따라 선택 정의할 수 있다.

5.4.4.3.AfterThrowing Advice

after() throwing()을 이용하여 AfterThrowing Advice를 정의한다. 다음은 PrintStringUsingAspectJ의 AfterThrowing Advice 정의 부분으로 해당 Pointcut 실행시 발생한 Exception 객체를 exception이라는 변수에 담도록 정의하고 있다. AfterThrowing Advice는 앞서 정의한 Pointcut에서 Exception이 발생한 후에 , "AfterThrowing Advice of PrintStringUsingAspectJ"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```
after() throwing(Exception exception) : getMethods() {
```



```

Class targetClass = thisJoinPoint.getTarget().getClass();
Signature signature = thisJoinPoint.getSignature();
String opName = signature.getName();

System.out.println("AfterThrowing Advice of PrintStringUsingAspctJ");
System.out.println("****" + targetClass + "." + opName + "()" + "****");
}

```

위에서 제시한 AfterThrowing Advice는 내부 정의된 JoinPoint 유형의 thisJoinPoint라는 객체를 이용하여, Target 클래스명, 메소드명 등과 같은 Target 정보를 추출하고 있다. 또한, 1개의 입력 인자(Exception)를 가지고 있는데 이것은 Pointcut 실행시 발생한 Exception 객체이다. AfterThrowing Advice에서 특정 Pointcut 실행시 발생한 Exception을 참조해야 한다면, Advice 정의시 throwing에 해당하는 객체를 메소드 로직 내에서 이를 활용하면 된다. 입력 인자는 AfterThrowing Advice 정의시 필요에 따라 선택 정의할 수 있다.

5.4.4.4.After(finally) Advice

after()를 이용하여 After(finally) Advice를 정의한다. 다음은 PrintStringUsingAspctJ 의 After(finally) Advice 정의 부분이다. After(finally) Advice는 앞서 정의한 getMethods()라는 Pointcut 후에 "After(finally) Advice of PrintStringUsingAspctJ"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```

after() : getMethods() {
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("After(finally) Advice of PrintStringUsingAspctJ");
    System.out.println("****" + targetClass + "." + opName + "()" + "****");
}

```

위에서 제시한 After(finally) Advice는 내부에 정의된 JoinPoint 유형의 thisJoinPoint라는 객체를 이용하여, Target 클래스명, 메소드명 등과 같은 Target 정보를 추출하고 있다.

5.4.4.5.Around Advice

around()를 이용하여 Around Advice를 정의한다. 다음은 PrintStringAroundUsingAspctJ 의 Around Advice 정의 부분으로 다른 Advice와 다르게 Return Type 정의가 추가되어 있음을 알 수 있다. Around Advice는 updateMethods()라는 Pointcut 후에 "Around Advice of PrintStringUsingAnnotation"라는 문자열과 해당 Pointcut을 가진 클래스명, 메소드명을 출력하는 역할을 수행한다.

```

Object around() : updateMethods() {
    Class targetClass = thisJoinPoint.getTarget().getClass();
    Signature signature = thisJoinPoint.getSignature();
    String opName = signature.getName();

    System.out.println("Around Advice of PrintStringUsingAspctJ");
    System.out.println("****" + targetClass + "." + opName + "()" + "****");
    // before logic
    Object retVal = proceed();
    // after logic
    return retVal;
}

```

위에서 제시한 Around Advice는 내부에 정의된 JoinPoint 유형의 thisJoinPoint라는 객체를 이용하여, Target 클래스명, 메소드명 등과 같은 Target 정보를 추출하고 있다. 또한 Around Advice 내에서 proceed()라는 메소드 호출을 통해 대상 Pointcut을 실행할 수 있어, Pointcut 전, 후 처리가 가능하며, Pointcut 실행 시점을 결정할 수 있게 된다. 또한 다른 Advice와는 달리 입력값, target, return 값 등에 대해 변경이 가능하다.

5.5.AOP Examples

다양한 부분에 Aspect을 적용할 수 있다. 이 페이지를 통하여 각 적용 예를 살펴보고, 적용 방법을 상세히 소개하고자 한다. 상세한 내용을 알고자 한다면, 아래 나열된 각 항목에 대한 링크를 참고하도록 한다.

5.5.1.AOP Example - Logging

개발된 어플리케이션 테스트시 오류가 발생한 경우, 해당하는 메소드 로직 내에 입력값 확인을 위해 DEBUG 레벨의 로그를 추가하거나, System.out.println() 구문을 추가하게 되는데 이로 인해 핵심 비즈니스 로직과 섞이게 되어 코드 복잡도가 증가한다. 따라서 특정 메소드 호출시 전달하는 입력값 확인을 위한 별도 Aspect을 정의하여 활용하면 관련된 메소드 내에 입력값 확인을 위한 로직들을 제외시킬 수 있게 된다. 다음에서는 AOP의 대표적인 툴 중 @AspectJ(Annotation)를 이용하여 Logging Aspect를 생성하고 테스트해 보도록 할 것이다. Logging Aspect 적용 대상은 GenericService 이며, 모든 메소드 실행 전에 해당 메소드를 실행하기 위해 입력된 인자들의 값을 로그로 남기는 역할을 수행하게 될 것이다.

5.5.1.1.Configuration

@AspectJ(Annotation)이 적용된 클래스들을 로딩하여 해당 클래스에 정의된 Pointcut, Advice를 실행하기 위해서는 Spring 속성 정의 XML 파일에 다음과 같이 추가해주어야 한다.

```
<aop:aspectj-autoproxy/>
```

log 출력을 위하여 log4j.xml 의 설정 내용을 확인하고 적절히 수정해 주어야 한다. 이번 예제에서는 org.anyframe.sample 패키지를 기준으로 로그를 출력할 예정이므로, 다음과 같이 DEBUG 레벨의 로거를 설정하여 준다.

```
<logger name="org.anyframe.sample" additivity="false">
  <level value="DEBUG"/>
  <appender-ref ref="console"/>
</logger>
```

5.5.1.2.Aspect 정의

다음과 같이 Annotation과 함께 구성된 LoggingAspect라는 Aspect 클래스를 생성한다. LoggingAspect라는 GenericService로 끝나는 클래스의 모든 메소드 실행 전에 해당 메소드 정보와 입력 인자값을 로그로 남기는 역할을 수행한다.

```
@Aspect
@Service
public class LoggingAspect {

    @Before("execution(* org.anyframe.sample.*Impl.*(..))")
    public void beforeLogging(JoinPoint thisJoinPoint) {
        Class<? extends Object> clazz = thisJoinPoint.getTarget().getClass();
        String methodName = thisJoinPoint.getSignature().getName();
        Object[] arguments = thisJoinPoint.getArgs();

        StringBuilder argBuf = new StringBuilder();
        StringBuilder argValueBuf = new StringBuilder();
        int i = 0;
        for (Object argument : arguments) {
            String argClassName = argument.getClass().getSimpleName();
            if (i > 0) {
                argBuf.append(", ");
            }
        }
    }
}
```

```

        argBuf.append(argClassName + " arg" + ++i);
        argValueBuf.append(".arg" + i + " : " + argument.toString() + "\n");
    }

    if (i == 0) {
        argValueBuf.append("No arguments\n");
    }

    StringBuilder messageBuf = new StringBuilder();
    messageBuf.append("before executing {} ({} method");

    messageBuf.append("\n-----\n");
    messageBuf.append(" {}");
    messageBuf.append("-----");

    Log logger = LoggerFactory.getLogger(clazz);
    logger.debug(
        messageBuf.toString(),
        new Object[]{methodName, argBuf.toString(), argValueBuf.toString()});
    }
}

```

beforeLogging() 메소드에서는 JoinPoint라는 객체를 이용하여, 해당 메소드 정보와 입력 인자값을 Target 클래스의 로를 통해 로그를 남기고 있음을 알 수 있다.

5.5.1.3.Aspect 실행

movieService, movieFinder를 호출하여, Movie CRUD 로직으로 구성된 Main.java 클래스를 실행시키면 Logging Aspect가 적용되어, 콘솔창을 통해 다음과 같은 형태의 로그를 볼 수 있다.

```

[DEBUG] 2011-04-18 13:40:42
org.anyframe.sample.aop.moviefinder.service.impl.MovieServiceImpl
before executing create(Movie arg1) method
-----
.arg1 : [actors=Shrek, director=Andrew Adamson, movieId=MV-99999, nowPlaying=N,
posterFile=null, releaseDate=Mon Apr 18 13:40:42 KST 2011, runtime=90, ticketPrice=8000.0,
title=Shrek (2001), genreId=GR-03]
-----
===== MovieDao : call create()=====

[DEBUG] 2011-04-18 13:40:42
org.anyframe.sample.aop.moviefinder.service.impl.MovieFinderImpl
before executing getPagingList(Movie arg1, Integer arg2) method
-----
.arg1 : [actors=Shrek, director=Andrew Adamson, movieId=MV-99999, nowPlaying=N,
posterFile=null, releaseDate=Mon Apr 18 13:40:42 KST 2011, runtime=90, ticketPrice=8000.0,
title=Shrek (2001), genreId=GR-03]
.arg2 : 1
-----
===== MovieDao : call getPagingList()=====

[DEBUG] 2011-04-18 13:40:42
org.anyframe.sample.aop.moviefinder.service.impl.MovieServiceImpl
before executing update(Movie arg1) method
-----
.arg1 : [actors=Shrek, director=Andrew Adamson, movieId=MV-99999, nowPlaying=N,
posterFile=null, releaseDate=Mon Apr 18 13:40:42 KST 2011, runtime=90, ticketPrice=8000.0,
title=Mission Impossible, genreId=GR-03]
-----
===== MovieDao : call update()=====

```

```
[DEBUG] 2011-04-18 13:40:42
org.anyframe.sample.aop.moviefinder.service.impl.MovieServiceImpl
before executing get(String arg1) method
-----
.arg1 : MV-99999
-----
===== MovieDao : call update()=====

[DEBUG] 2011-04-18 13:40:42
org.anyframe.sample.aop.moviefinder.service.impl.MovieServiceImpl
before executing remove(String arg1) method
-----
.arg1 : MV-99999
-----
===== MovieDao : call remove()=====
```

5.5.2.AOP Example - Exception Transfer

특정 비즈니스 로직 수행시 발생할 수 있는 Exception에 대한 로그 및 메시지 처리를 수행하기 위해 핵심 비즈니스 로직외에 Exception 처리 로직이 추가되어야 한다. 때문에 핵심 비즈니스 로직외에 매 로직마다 반복되는 try ~ catch 블록으로 인해 코드가 복잡해진다. 만일 별도 Aspect을 통해 공통적으로 Exception들을 처리하게 하고, 각 비즈니스 로직에서 try ~ catch 블록을 제거할 수 있다면 코드가 훨씬 간단해지고, 궁극적으로 개발자는 비즈니스 로직에만 집중할 수 있는 기반이 마련될 수 있을 것이다. 다음에서는 AOP 대표적인 틀 중 Spring AOP를 이용하여 XML 스키마 기반에서 ExceptionTransfer를 위한 Aspect를 생성하고 테스트해 보도록 할 것이다. ExceptionTransfer Aspect 적용 대상은 GenericService로 끝나는 클래스의 모든 메소드 실행시 Exception이 발생한 경우 이를 처리하기 위한 역할을 수행하게 될 것이다.

5.5.2.1.Aspect 정의

Spring 속성 정의 XML(context.xml.xml) 파일 내에 Aspect 클래스를 Bean으로 정의한 후, 해당 Aspect에 대한 Pointcut과 Advice를 정의한다.

```
<aop:config>
  <aop:pointcut id="serviceMethod" expression="execution(execution(*
org.anyframe.sample.*Impl.*(..)))" />
  <aop:aspect ref="exceptionTransfer" order="1">
    <aop:after-throwing throwing="exception" pointcut-ref="serviceMethod"
method="transfer" />
  </aop:aspect>
</aop:config>
```

ExceptionTransfer는 Impl로 끝나는 클래스의 모든 메소드 실행시 발생한 Exception에 대해 처리하는 역할을 수행한다.

5.5.2.2.Advice 구현

다음과 같이 ExceptionTransfer 라는 Aspect 클래스를 생성한다.

```
public class ExceptionTransfer {
    private MessageSource messageSource;

    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
}
```

```

public void transfer(JoinPoint thisJoinPoint, Exception exception)
    throws MovieFinderException {
    Object target = thisJoinPoint.getTarget();
    while (target instanceof Advised) {
        try {
            target = ((Advised) target).getTargetSource().getTarget();
        } catch (Exception e) {
            LoggerFactory.getLogger(this.getClass()).error(
                "Fail to get target object from JointPoint.", e);
            break;
        }
    }

    String className = target.getClass().getSimpleName().toLowerCase();
    String opName = (thisJoinPoint.getSignature().getName()).toLowerCase();
    Log logger = LoggerFactory.getLogger(target.getClass());

    if (exception instanceof MovieFinderException) {
        MovieFinderException empEx = (MovieFinderException) exception;
        logger.error(empEx.getMessage(), empEx);
        throw empEx;
    }

    try {
        logger.error(messageSource.getMessage("error." + className + "."
            + opName, new String[] {}, Locale.getDefault()), exception);
    } catch (Exception e) {
        logger.error(messageSource.getMessage("error.common",
            new String[] {}, Locale.getDefault()), exception);
        throw new MovieFinderException(messageSource, "error.common");
    }
    throw new MovieFinderException(messageSource, "error." + className
        + "." + opName);
}
}

```

transfer() 메소드에서는 발생한 Exception 객체의 유형을 MovieFinderException, 기타로 구분하고 Exception이 발생한 Target 클래스와 메소드명을 조합한 message key를 이용하여 해당하는 메시지를 얻어낸다. 그런 후에 이 메시지를 이용하여 ERROR 레벨의 로그를 남긴 후에 MovieFinderException으로 전환하여 throw한다.

5.5.3.AOP Example - Profiler

별도 성능 측정 틀없이도 Aspect을 통해 응답 속도가 중요시 되는 일부 메소드에 대해 개발 시점에 미리 메소드 수행에 걸리는 시간을 측정해 볼 수 있다. 따라서 개발시에 성능 저하의 요인이 되는 지점을 미리 파악하고 대처해 볼 수 있을 것이다.

다음에서는 AOP의 대표적인 툴 중 @AspectJ(Annotation)를 이용하여 SimpleProfiler Aspect를 생성하고 테스트해 보도록 할 것이다. SimpleProfiler Aspect 적용 대상은 UserService이며, 특정 메소드(add*) 실행에 소요되는 시간을 측정하고, 이를 콘솔에 남기는 역할을 수행하게 될 것이다.

5.5.3.1.Configuration

@AspectJ(Annotation)이 적용된 클래스들을 로딩하여 해당 클래스에 정의된 Pointcut, Advice를 실행하기 위해서는 Spring 속성 정의 XML 파일에 다음과 같이 추가해주어야 한다.

```
<aop:aspectj-autoproxy/>
```

5.5.3.2.Aspect 정의

다음과 같이 Annotation과 함께 구성된 SimpleProfiler 라는 Aspect 클래스를 생성한다. SimpleProfiler 는 org.anyframe.sample 패키지 내에 속한 모든 클래스 중 클래스명이 Impl로 끝나는 모든 클래스 내 의 메소드명이 create로 시작하는 메소드를 대상으로 한다. 그리고 해당 메소드의 실행 전후에 Spring 에서 제공하는 Stopwatch를 이용하여 메소드 실행에 소요되는 시간을 측정하고, 이를 콘솔에 남기는 역할을 수행하게 될 것이다.

```
@Aspect
public class SimpleProfiler {
    @Around("execution(* org.anyframe.sample.aop.movidfinder.service.*Impl.create*(..))")
    public Object profile(ProceedingJoinPoint thisJoinPoint) throws Throwable {
        String className = thisJoinPoint.getSignature().getDeclaringTypeName();
        Stopwatch stopwatch = new Stopwatch("Profiling for [" + className + "]");
        try {
            stopwatch.start(thisJoinPoint.toShortString());
            return thisJoinPoint.proceed();
        } finally {
            stopwatch.stop();
            System.out.println(stopwatch.shortSummary());
        }
    }
}
```

5.5.3.3.Aspect 실행

UserService를 호출하여, 구현 로직에서 1000 milliseconds 동안 멈추도록 로직이 추가되어 있는, 신규 User 정보 등록 기능을 호출하는 SimpleProfilerAspectTest 클래스를 실행시키면 SimpleProfiler Aspect 가 적용되어, 콘솔창을 통해 다음과 같은 형태의 로그를 볼 수 있다.

```
StopWatch 'Profiling for
[org.anyframe.sample.aop.movielfinder.service.impl.MovieServiceImpl]'
: running time (millis) = 1016
```

5.5.4.AOP Example - Design Level Assertions

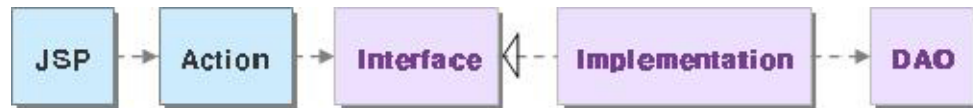
개발 표준이라 함은 각종 명명 표준 및 해당 프로젝트에서 기 검증한 소프트웨어 아키텍처 스타일에 맞춰 개발 작업을 수행할 수 있도록 가이드한다. 따라서, 개발자들이 개발 초기에 겪게 되는 혼선을 줄이고 비즈니스 로직에만 집중할 수 있도록 하며, 동일한 표준을 준수한 코드에 대해서는 유지보수 및 변경이 용이하다. 대부분의 프로젝트에서는 어플리케이션을 본격적으로 개발하기에 앞서 상당한 시간을 들여 해당 프로젝트에 적합한 개발 표준을 별도 문서로 정의하고 개발자들이 이를 준수하여 개발 작업을 수행할 것을 권장하나, 제대로 지켜지지 않고 있으며 이에 대한 검증 또한 한계가 있는게 사실이다.

만일 코드 컴파일시에 개발 표준을 적용할 수 있다면, 코딩시에 손쉽게 표준에 부적합한 코드를 인식하고 수정할 수 있게 될 것이다. 이를 위해 본 문서에서는 Design Rule을 declare error/warning 문으로 구성된 Aspect로 정의하고, 부적합 사항을 찾아 수정하는 방법에 대해 알아보기로 하자.

먼저, declare error/warning 문은 다음과 같이 정의하며, Pointcut Expression에 해당하는 JoinPoint가 있을 경우 정의된 메시지를 보여준다.

```
@DeclareWarning("Pointcut Expressions")
static final String variableName = "msg...";
```

다음은 Anyframe 기반 어플리케이션 개발시 가장 흔하게 볼 수 있는 일부 소프트웨어 아키텍처 그림이다.



해당 어플리케이션의 패키지는 com.sds.emp로 시작하며, 프리젠테이션 레이어는 com.sds.emp.서브 모듈명.web, 비즈니스 레이어는 com.sds.emp.서브모듈명.service 내에 위치한다라고 가정하자.

정의 가능한 Design Rule은 크게 Interaction Rule, Naming Rule로 구분해 볼 수 있으며, 이제부터 위 그림을 기반으로 Design Rule을 하나씩 정의해 보도록 하자.

5.5.4.1. Interaction Rule 정의 예제

패키지 레벨, 클래스 레벨 등에서 필요한 Pointcut을 정의하고 Declare 문에서 앞서 정의한 여러 Pointcut을 조합하여 클래스간 Interaction Rule을 정의하였다. 이는 기 정의된 Pointcut을 다른 Declare 문에서 재사용하기 위함이다. 다음은 DevStandard Aspect에 정의된 Interaction Rule의 일부이다.

1. 프리젠테이션 레이어에 속하지 않은 클래스에서 **Action** 또는 **Form** 클래스를 호출할 수 없다.

```

// 패키지명인 com.sds.emp로 시작하고 중간에 web을 포함하는 모든 패키지에 속한 JoinPoint
@Pointcut("within(com.sds.emp..web..*)")
public void inWebPkg() {}

// 클래스명인 Action 또는 Form으로 끝나는 클래스의 모든 메소드 호출하는 JoinPoint
@Pointcut("call(* com.sds.emp..web.*Action.*(..)) && call(* com.sds.emp..web.*Form.*(..))")
public void callToWeb() {}

// web 패키지에 속하지 않은 클래스에서 web 패키지 내의 Action 또는 Form 클래스를 호출하는 경우
// 다음과 같은 Error 메시지를 보여준다.
@DeclareError("!inWebPkg() && callToWeb()")
static final String irMsg5 = "web 패키지에 속한 모든 클래스에 접근할 수 없습니다.";
  
```

2. 프리젠테이션 레이어에서는 반드시 **Interface**를 통해 특정 서비스에 접근해야 한다.

```

// 패키지명인 com.sds.emp로 시작하고 중간에 web을 포함하는 모든 패키지에 속한 JoinPoint
@Pointcut("within(com.sds.emp..web..*)")
public void inWebPkg() {}

// 클래스명인 DAO로 끝나는 클래스의 모든 메소드 호출하는 JoinPoint
@Pointcut("call(* com.sds.emp..service.impl.*DAO.*(..))")
public void callToDAO() {}

// 클래스명인 Impl로 끝나는 클래스의 모든 메소드 호출하는 JoinPoint
@Pointcut("call(* com.sds.emp..service.impl.*Impl.*(..))")
public void callToImplementation() {}

// web 패키지에서 DAO 또는 Impl 내의 메소드를 직접 호출하는 경우 다음과 같은 Error 메시지를
// 보여준다.
@DeclareError("inWebPkg() && ( callToDAO() || callToImplementation())")
static final String irMsg1 = "Action 클래스에서는 특정 서비스의 구현 클래스나 DAO 클래스에
직접 "
+ "접근할 수 없습니다.";
  
```

3. 특정 객체(java.sql.Connection)를 직접 사용하지 않도록 한다.

```

// 패키지명인 integration 또는 unit으로 시작하는 모든 테스트 패키지 속한 JoinPoint
@Pointcut("within(integration..* || unit..*)")
public void inTestPkg() {}
  
```

```
// java.sql.Connection 클래스의 모든 메소드를 호출하는 JoinPoint
@Pointcut("call(* java.sql.Connection.*(..))")
public void callToConnection() {}

// 테스트 패키지를 제외한 모든 패키지에서 Connection 객체를 직접 호출하는 경우 다음과 같은
// Error 메시지를 보여준다.
@DeclareError("callToConnection() && !inTestPkg()")
static final String irMsg2 = "java.sql.Connection 객체에 직접 접근할 수 없습니다. "
+ "Anyframe Service를 이용하세요.";
```

4. 생성자를 직접 호출하여 **DAO** 인스턴스를 생성할 수 없다. **Dependency Injection**을 통해 객체간 참조 관계를 정의해야 한다.

```
// DAO 클래스의 Constructor를 호출하는 JoinPoint
@Pointcut("call(com.sds.emp..service.impl.*DAO.new(..))")
public void callToDAOConstructor() {}

// DAO 클래스를 Constructor를 직접 호출하는 경우 다음과 같은 Error 메시지를 보여준다.
@DeclareError("callToDAOConstructor()")
static final String irMsg3 = "DAO 인스턴스를 직접 생성하실 수 없습니다. "
+ "객체간 참조 관계는 서비스 속성 정의 XML에 정의하여 사용하세요.";
```

5.5.4.2.Naming Rule 정의 예제

패키지 레벨, 클래스 레벨 등에서 필요한 Pointcut을 정의하고 Declare 문에서 앞서 정의한 여러 Pointcut을 조합하여 Naming Rule을 정의하였다. 이는 기 정의된 Pointcut을 다른 Declare 문에서 재사용하기 위함이다. 다음은 DevStandard Aspect에 정의된 Naming Rule의 일부이다.

1. **com.sds.emp.서브모듈명.web** 패키지 내에 존재하는 클래스명은 **Action** 또는 **Form**으로 끝나야 한다.

```
// 패키지명이 com.sds.emp로 시작하고 중간에 web을 포함하는 모든 패키지에 속한 JoinPoint
@Pointcut("within(com.sds.emp..web..*)")
public void inWebPkg() {}

// 메소드나 Constructor, 메소드 토직이 아닌 모든 JoinPoint 즉, 클래스 정의 부분만 해당
@Pointcut("!(execution(* *(..)) || withincode(*.new(..)) || withincode(* *(..)))")
public void clazz(){}

// 패키지명이 com.sds.emp로 시작하고 중간에 web을 포함하는 모든 패키지에 속한 JoinPoint 중
// 클래스명이 Action으로 끝나는 클래스에 속한 모든 JoinPoint
@Pointcut("within(com.sds.emp..web..*Action)")
public void actionName() {}

// 패키지명이 com.sds.emp로 시작하고 중간에 web을 포함하는 모든 패키지에 속한 JoinPoint 중
// 클래스명이 Form으로 끝나는 클래스에 속한 모든 JoinPoint
@Pointcut("within(com.sds.emp..web..*Form)")
public void formName() {}

// web 패키지에 속하면서 클래스명이 Action이나 Form으로 끝나지 않는 클래스 정의 부분이 있을
// 경우,
// 다음과 같은 warning 메시지를 보여준다.
@DeclareWarning ("inWebPkg() && clazz() && !(actionName() || formName())")
static final String nrMsg2 = "web 패키지에 속한 모든 클래스의 이름은
Action 또는 Form으로 끝나야 합니다.";
```

2. **com.sds.emp.서브모듈명.service.impl** 패키지 내에 존재하는 클래스명은 **Impl** 또는 **DAO**로 끝나야 한다.


```
// 패키지명이 com.sds.emp로 시작하고 중간에 service.impl을 포함하는 모든 패키지에 속한
JoinPoint
@Pointcut("within(com.sds.emp..service.impl..*)")
public void inImplementationPkg() {}

// 메소드나 Constructor, 메소드 토직이 아닌 모든 JoinPoint 즉, 클래스 정의 부분만 해당
@Pointcut("!(execution(* *(..)) || withincode(* *(..)) || withincode(*.new(..)))")
public void clazz(){}

// 패키지명이 com.sds.emp로 시작하고 중간에 service.impl을 포함하는 모든 패키지에 속한
JoinPoint 중
// 클래스명이 Impl로 끝나는 클래스에 속한 모든 JoinPoint
@Pointcut("within(com.sds.emp..service.impl..*Impl)")
public void implementationName() {}

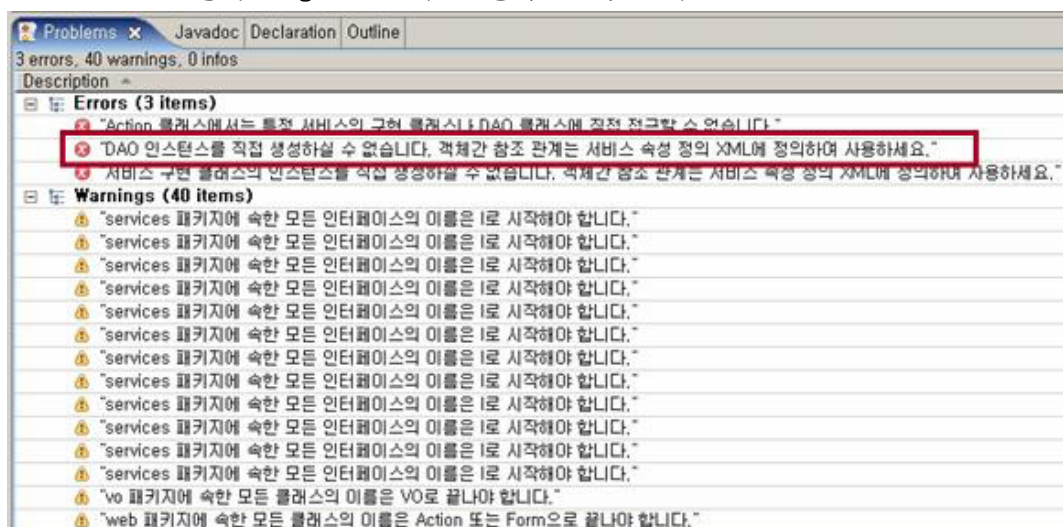
// 패키지명이 com.sds.emp로 시작하고 중간에 service.impl을 포함하는 모든 패키지에 속한
JoinPoint 중
// 클래스명이 DAO로 끝나는 클래스에 속한 모든 JoinPoint
@Pointcut("within(com.sds.emp..service.impl..*DAO)")
public void daoName() {}

// services.impl 패키지에 속하면서 클래스명이 Impl이나 DAO로 끝나지 않는 클래스 정의 부분이
있을 경우,
// 다음과 같은 warning 메시지를 보여준다
@DeclareWarning ("inImplementationPkg() && clazz() && !(implementationName() ||
daoName())")
static final String nrMsg4 = "services 패키지에 속한 모든 클래스의 이름은
Impl 또는 DAO로 끝나야 합니다.";
```

5.5.4.3.Refactoring

Eclipse 기반하에 어플리케이션을 개발하면 Design Rule을 위반한 코드를 보다 손쉽게 수정할 수 있다.

1. Eclipse 작업 공간 내에 Problems View가 없다면, Eclipse 메뉴 Window > Show View > Problems를 선택하여 Problems view를 오픈한다.
2. Problems View를 통해 Design Rule을 위반한 항목들을 확인한다.



3. Problems View에서 수정할 항목을 더블 클릭하여 대상 코드로 이동한다.



```

public Page getUserList(SearchVO searchVO) throws EmpException {
    UserDAO dao2 = new UserDAO();
    try {
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug(messageSource.getMessage(
                "debug.user.get.list.condition",
                new String[] { searchVO.getSearchCondition() }, Locale
                    .getDefault()));
            LOGGER.debug(messageSource.getMessage(
                "debug.user.get.list.keyword", new String[] { searchVO
                    .getSearchKeyword() }, Locale.getDefault()));
            LOGGER.debug(messageSource.getMessage(
                "debug.user.get.list.pageIndex",
                new String[] { searchVO.getPageIndex() + "" }, Locale
                    .getDefault()));
        }
        return dao2.getUserList(searchVO);
    }
}

```

4. Design Rule을 준수한 코드로 수정함으로써 Problem을 제거한다.

```

public void setUserDAO(UserDAO userDAO) {
    this.userDAO = userDAO;
}

public Page getUserList(SearchVO searchVO) throws EmpException {
    //UserDAO dao2 = new UserDAO();
    try {
        // 중략
        return userDAO.getUserList(searchVO);
    }
    // 중략
}

```

5.6.Resources

- 참고자료
- ZDNet Korea의 제휴 매체인 마이크로소프트웨어 - AOP [<http://dev.anyframejava.org/anyframe/doc/core/3.2.1/downloads/corefw/guide/aop/AOP-Full.doc>]

6.SpEL(Spring Expression Language)

Spring 3에서 새롭게 선보이고 있는 Spring Expression Language(이하 SpEL)는 Expression Language의 하나로써 런타임시 특정 객체의 정보에 접근하거나 조작할 수 있도록 지원한다. Syntax는 Unified EL과 유사하나 부가적인 기능을 추가 제공하고 있다. 또한 SpEL는 Spring의 모든 내부 프로젝트들의 Expression Evaluation을 위한 기반으로써 활용된다. 본 섹션에서는 XML/Annotation 기반으로 Bean 정의시 SpEL을 어떻게 활용하는지, Spring에서 제공하는 Expression Interface를 직접 이용하여 정의된 Expression을 어떻게 Evaluation하는지 알아보도록 하자. 그리고 Spring 기반에서 Expression 정의를 위한 Syntax는 Language Reference를 참고하도록 한다.

6.1.Bean Definition using SpEL

Spring에서 XML/Annotation 기반으로 Bean을 정의할 때 #{expression} 형태로 SpEL를 활용할 수 있다.

6.1.1.XML based Bean Definition

다음은 Core Plugin 설치로 추가된 Spring 속성 정의 파일 context-transaction.xml의 일부로써 SpEL을 활용하여 dataSource Bean을 정의하고 있다. 클래스패스 상에 존재하는 context.properties 파일을 로드하여 contextProperties라는 이름의 Bean에서 관리하도록 정의해두고, dataSource Bean에서는 contextProperties Bean에 접근하여 driver, url, username, password에 대한 속성 정보를 추출함으로써 해당 Bean의 속성을 셋팅하고 있음을 알 수 있다.

```
<util:properties id="contextProperties" location="classpath:context.properties"/>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName" value="#{contextProperties.driver}"/>
    <property name="url" value="#{contextProperties.url}"/>
    <property name="username" value="#{contextProperties.username}"/>
    <property name="password" value="#{contextProperties.password}"/>
</bean>
```

'systemProperties'는 SpEL에서 정의한 내부 변수명으로 시스템 변수의 값을 추출하고자 할 때 #{systemProperties[...] }와 같은 형태로 활용할 수 있다. 즉, 앞서 언급한 DB 속성 정보를 Driver, URL, UserName, Password라는 시스템 변수로 셋팅해 두었다면 다음과 같이 dataSource Bean의 속성 정의를 변경할 수 있을 것이다.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName" value="#{systemProperties['Driver']}"/>
    <property name="url" value="#{systemProperties['URL']}"/>
    <property name="username" value="#{systemProperties['UserName']}"/>
    <property name="password" value="#{systemProperties['Password']}"/>
</bean>
```

6.1.2.Annotation based Bean Definition

Spring에서 Annotation 기반으로 Bean을 정의할 때, XML과 달리 @Value라는 Annotation과 함께 Expression을 정의해 주어야 하며 @Value는 field, method, method/constructor argument에 적용할 수 있다.

다음은 Core Plugin 설치로 추가된 DAO 클래스 ~/service/impl/MovieDao.java의 일부로써 SpEL을 활용하여 MovieDao Bean의 pageSize, pageUnit 속성을 셋팅하고 있음을 알 수 있다. contextProperties

Bean으로부터 'pageSize', 'pageUnit' 정보를 추출하여 해당하는 field의 값을 셋팅하고 이를 활용하게 될 것이다.

```
@Repository("coreMovieDao")
public class MovieDao extends JdbcDaoSupport {
    @Value("#{contextProperties['pageSize'] ?: 10}")
    int pageSize;

    @Value("#{contextProperties['pageUnit'] ?: 10}")
    int pageUnit;

    @Inject
    public void setJdbcDaoDataSource(DataSource dataSource) {
        super.setDataSource(dataSource);
    }

    ...
}
```

위 코드에서는 Elvis Operator를 활용하여 Expression을 정의하고 있으며 contextProperties라는 이름의 Bean이 'pageSize' 값을 가지고 있지 않을 경우 기본값을 '10'으로 셋팅하도록 요구하고 있다.

Annotation 기반 Bean 정의시에도 XML과 마찬가지로 'systemProperties'를 활용하여 시스템 변수의 값을 추출할 수 있다.

6.2.Expression Evaluation using Spring's Expression Interface

org.springframework.expression.ExpressionParser는 정의된 Expression을 Parsing하여 org.springframework.expression.Expression 형태로 리턴한다. 그리고 org.springframework.expression.Expression은 앞서 Parsing한 Expression을 Evaluating하는 역할을 수행하게 된다.

```
ExpressionParser parser = new SpelExpressionParser();
Expression expression = parser.parseExpression("'Hello Anyframe'");
String stringVal = (String)expression.getValue();
```

따라서 위의 코드 실행 결과 Expression 객체는 정의된 Literal Expression "'Hello Anyframe'"을 Evaluate하여 'stringVal'에 'Hello Anyframe'라는 값을 할당하게 될 것이다. Evaluate된 결과를 특정 타입의 클래스로 전달받고자 하는 경우에는 getValue() 메소드 호출시 인자로 클래스 타입을 정의해줄 수 있다.

```
ExpressionParser parser = new SpelExpressionParser();
Expression expression = parser.parseExpression("'Hello Anyframe'");
String stringVal = expression.getValue(String.class);
```

SpEL에서는 정의된 Expression을 Evaluate하여 특정 객체의 속성 정보에 접근하거나 수정하기 위해 2가지 방법을 제공한다. Expression을 통해 Movie 객체로부터 'nowPlaying' 속성값에 접근하는 로직을 기반으로 각 방법에 대해 알아보도록 하자.

다음 코드에서는 Expression의 getValue() 메소드 호출시 Movie 객체를 셋팅한 org.springframework.expression.spel.support.StandardEvaluationContext 객체를 활용하고 있다. 따라서 EvaluationContext를 기반으로 Expression Evaluation이 이루어지게 될 것이다. 그러나 Evaluation 대상이 되는 객체가 자주 변경되어야 하는 경우에는 해당 객체가 변경될 때마다 SpelExpressionParser() 생성 로직이 매번 구현되어야 하므로 적절치 않은 방법이다.

```
Movie movie = getGenre().getMovies().get(0);
```

```
StandardEvaluationContext context = new StandardEvaluationContext(movie);

ExpressionParser parser = new SpelExpressionParser();
String playing = parser.parseExpression("nowPlaying").getValue(context, String.class);
```

다음은 Expression의 `getValue()` 메소드 호출시 Movie 객체를 직접 활용하고 있다. 이 경우 Expression Evaluation할 때마다 내부적으로 EvaluationContext가 새롭게 생성되어 관리될 것이다. Evaluation 대상이 되는 객체가 자주 변경되어야 하는 경우에 활용할 수 있는 방법이다.

```
Movie movie = getGenre().getMovies().get(0);
ExpressionParser parser = new SpelExpressionParser();
String playing = parser.parseExpression("nowPlaying").getValue(movie, String.class);
```

단, SpEL에서는 다음과 같은 이유로 첫번째 언급한 방법을 사용할 것을 권장하고 있다.

- EvaluationContext를 생성하고 구성하는데 소요되는 비용이 상대적으로 비싸다.
- EvaluationContext가 상태 정보를 캐싱함으로써 재사용시 다음 Expression Evaluation이 보다 빠르게 수행될 수 있도록 지원한다.

6.3.Language Reference

다음에서는 다양한 형태의 Spring Expression 정의 방법에 대해 예제 코드와 함께 자세히 알아보도록 하자.

6.3.1.Literal Expressions

정의된 문자열을 String, Date, Number, boolean, null 타입으로 변경하여 전달할 수 있다. String 타입의 경우 single quote(')를 사용하여 정의하도록 한다. 다음은 Spring ExpressionParser를 활용하여 다양한 문자열을 지정된 타입으로 변경해보는 로직을 포함한 테스트 메소드이다.

```
public void evaluateLiteralExpression() throws Exception {
    ExpressionParser parser = new SpelExpressionParser();
    String stringVal = (String) parser.parseExpression("'Hello Anyframe'").getValue();
    System.out.println("[Literal Expression] Evaluate 'Hello Anyframe' : " + stringVal);

    double doubleVal = (Double) parser.parseExpression("6.0221415E+23").getValue();
    System.out.println("[Literal Expression] Evaluate '6.0221415E' : " + doubleVal);

    Date dateVal = (Date) parser.parseExpression("'2010/07/05'").getValue(Date.class);
    System.out.println("[Literal Expression] Evaluate '2010/07/05' : " + dateVal);

    boolean booleanVal = (Boolean) parser.parseExpression("true").getValue();
    System.out.println("[Literal Expression] Evaluate 'true' : " + booleanVal);

    Object nullVal = parser.parseExpression("null").getValue();
    System.out.println("[Literal Expression] Evaluate null : " + nullVal);
}
```

위 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Literal Expression] Evaluate 'Hello Anyframe' : Hello Anyframe
[Literal Expression] Evaluate '6.0221415E' : 6.0221415E23
[Literal Expression] Evaluate '2010/07/05' : Mon Jul 05 00:00:00 KST 2010
[Literal Expression] Evaluate 'true' : true
[Literal Expression] Evaluate null : null
```

6.3.2.Properties, Arrays, Lists, Maps, Indexers

Properties, Arrays, List, Map 등과 같은 객체가 가진 내부 객체 목록에 접근하기 위해 ['index'] 또는 ['key']와 같은 형태의 Expression을 정의할 수 있다. 또한 period(.)를 사용하면 내부 객체의 속성 정보에 접근할 수 있게 된다. (특정 객체의 속성 정보에 접근시 속성명의 첫번째 문자에 대해서는 Case Insensitive하다.) 다음은 Genre 객체 내에 포함된 java.util.ArrayList 타입의 movies로부터 첫번째 Movie 정보의 Title 값을 추출해 보는 테스트 메소드이다.

```
public void evaluateListExpression() throws Exception {
    Genre genre = getGenre();
    StandardEvaluationContext context = new StandardEvaluationContext(genre);

    ExpressionParser parser = new SpelExpressionParser();
    String title = parser.parseExpression("movies[0].Title").getValue(context,
    String.class);
    System.out.println("[List Expression] Movie title : " + title);
}
```

위 메소드에서 활용한 테스트 데이터 Genre가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 첫번째 Movie 정보의 Title 값인 'Shrek (2001)'이 출력될 것이다.

```
[List Expression] Movie title : Shrek (2001)
```

6.3.3.Methods

일반적인 Java Syntax를 활용하여 지정된 메소드를 실행시킬 수 있다. 다음은 Spring ExpressionParser를 활용하여 substring()이라는 메소드를 실행하는 테스트 로직의 일부이다.

```
ExpressionParser parser = new SpelExpressionParser();
String c = parser.parseExpression("'abc'.substring(1)").getValue(String.class);
```

Expression이 위와 같이 정의된 경우 특정 문자열을 substring한 결과인 'bc'라는 결과값이 리턴될 것이다.

6.3.4.Relational Operators

equal('=='), not equal('!='), less than('<'), less than or equal('<='), greater than('>'), and greater than or equal('>=') 등과 같은 Relational Operator를 활용할 수 있다. XML 문서 등에서의 <와 같은 사용으로 인한 문제를 피하기 위해서는 Relational Operators는 문자로도 표현할 수 있다. (case insensitive)

```
lt ('<'), gt ('>'), le ('<='), ge ('>='),
eq ('=='), ne ('!='), div ('/'), mod ('%'), not ('!')
```

이 외에도 'instanceof'와 Regular Expression 기반의 'matches'도 지원한다. 다음은 Spring ExpressionParser를 활용하여 Relational Operator를 포함한 Expression을 실행하는 테스트 로직의 일부이다.

```
ExpressionParser parser = new SpelExpressionParser();
boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);
boolean falseValue = parser.parseExpression("'xyz' instanceof
T(int)").getValue(Boolean.class);
boolean trueValue =
    parser.parseExpression("'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);
```

6.3.5.Logical Operators

and, or, not 등과 같은 Logical Operator를 활용할 수 있다. 다음은 Spring ExpressionParser를 활용하여 Logical Operator를 포함한 Expression을 실행하는 테스트 로직의 일부이다.

```
ExpressionParser parser = new SpelExpressionParser();
boolean falseValue = parser.parseExpression("true and false").getValue(Boolean.class);
boolean trueValue = parser.parseExpression("true or false").getValue(Boolean.class);
boolean falseValue = parser.parseExpression("!true").getValue(Boolean.class);
```

6.3.6.Mathematical Operators

+, -, *, /, %, ^와 같은 Mathematical Operator를 활용할 수 있다. 다음은 Spring ExpressionParser를 활용하여 Mathematical Operator를 포함한 Expression을 실행하는 테스트 로직의 일부이다.

```
ExpressionParser parser = new SpelExpressionParser();
String testString =
    parser.parseExpression("'test' + ' ' + 'string'").getValue(String.class);
int two = parser.parseExpression("1 + 1").getValue(Integer.class);
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.class);
```

6.3.7.Assignment

Assignment Operator('=')를 활용하여 setter 메소드를 실행한 것과 동일하게 특정 객체의 속성값을 수정할 수 있다. 다음은 Assignment Operator를 활용하여 Genre 객체의 'Name' 값을 'Animation'으로 변경해보는 로직을 포함한 테스트 메소드이다.

```
public void evaluateAssignmentExpression() throws Exception {
    Genre genre = getGenre();
    StandardEvaluationContext context = new StandardEvaluationContext(genre);

    ExpressionParser parser = new SpelExpressionParser();
    String genreName = parser.parseExpression("Name = 'Animation'")
        .getValue(context, String.class);
    System.out.println("[Assignment Expression] Genre Name : " + genreName);
}
```

위 메소드에서 활용한 테스트 데이터 Genre가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 'Name' 값이 'Adventure'였으나 Assignment Operator를 활용하여 값을 변경하였으므로 위 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Assignment Expression] Genre Name : Animation
```

6.3.8.Types

특정 타입의 클래스를 표현하기 위해 'T' Operator를 활용할 수 있다. 또한 'T' Operator를 활용할 경우 특정 클래스의 static 메소드 호출이 가능하다. java.lang 패키지 하위의 클래스일 경우 패키지를 별도로 명시하지 않아도 무방하다. 다음은 Spring ExpressionParser를 활용하여 'T' Operator를 포함한 Expression을 실행하는 테스트 로직의 일부이다.

```
ExpressionParser parser = new SpelExpressionParser();
Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);
Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);
```

6.3.9. Constructors

'new' Operator를 활용하여 특정 클래스의 Constructor를 호출할 수 있다. 클래스명은 fully qualified 형태로 기술해줘야 한다. (단, primitive 타입의 클래스, java.lang 패키지의 클래스는 제외) 다음은 'new' Operator를 활용하여 Genre 클래스의 Constructor를 호출하여 신규 Genre 객체를 생성해보는 로직을 포함한 테스트 메소드이다.

```
public void evaluateConstructorExpression() throws Exception {
    ExpressionParser parser = new SpelExpressionParser();
    Genre genre = parser.parseExpression(
        "new org.anyframe.sample.domain.Genre('GR-01', 'Action')")
        .getValue(Genre.class);
    System.out.println("[Constructor Expression] Genre Id : "
        + genre.getGenreId() + ", Genre Name : " + genre.getName());
}
```

위 메소드 실행 결과 Constructor 호출로 인해 신규 Genre 객체가 생성되었을 것이며 이 Genre 객체는 GenreId : 'GR-01', GenreName : 'Action'이라는 값을 갖고 있을 것이다. 따라서 위 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Constructor Expression] Genre Id : GR-01, Genre Name : Action
```

6.3.10. Variables

StandardEvaluationContext의 setVariable() 메소드를 호출하여 변수를 정의하고, 정의된 변수를 #{변수명} 형태로 Spring Expression 내에서 활용할 수 있다. 다음은 'newName'이라는 변수의 값을 'Animation'으로 정의해두고 Genre 객체의 'Name' 속성값을 #newName을 활용하여 변경해보는 로직을 포함한 테스트 메소드이다.

```
public void evaluateVariablesExpression() throws Exception {
    Genre genre = getGenre();
    StandardEvaluationContext context = new StandardEvaluationContext(genre);
    context.setVariable("newName", "Animation");

    ExpressionParser parser = new SpelExpressionParser();
    parser.parseExpression("Name = #newName").getValue(context);
    System.out.println("[Variables Expression] Genre New Name : "
        + genre.getName());
}
```

위 메소드에서 활용한 테스트 데이터 Genre가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 'Name' 값이 'Adventure'였으나 #newName을 활용하여 값을 변경하였으므로 위 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Variables Expression] Genre New Name : Animation
```

단, #this와 #root는 예약어로서 #this는 현재 evaluation 대상이 되는 객체를, #root는 Context에 셋팅된 Root 객체를 의미한다.

6.3.11. Functions

StandardEvaluationContext의 registerFunction() 메소드를 호출하여 특정 클래스의 메소드를 등록하고, 해당 메소드를 #{메소드명} 형태로 Spring Expression 내에서 활용할 수 있다. 다음은 org.springframework.util.StringUtils 클래스 내의 'capitalize'라는 메소드를 Spring Expression내에서 호출해보는 로직을 포함한 테스트 메소드이다.


```
public void evaluateFunctionExpression() throws Exception {
    StandardEvaluationContext context = new StandardEvaluationContext();
    context.registerFunction("capitalize", StringUtils.class
        .getDeclaredMethod("capitalize", new Class[] { String.class }));

    ExpressionParser parser = new SpelExpressionParser();
    String capitalizedString = parser.parseExpression(
        "#capitalize('hello anyframe')")
        .getValue(context, String.class);
    System.out.println("[Function Expression] Capitalized String : "
        + capitalizedString);
}
```

위 메소드 실행 결과 capitalize() 메소드의 인자로 정의된 'hello anyframe'이라는 문자열에 대해 첫번째 문자가 Capitalize되어 다음과 같이 출력될 것이다.

```
[Function Expression] Capitalized String : Hello anyframe
```

6.3.12.Ternary Operator

if-then-else 로직 수행을 위해 Ternary Operator를 활용할 수 있다. 다음은 Movie 객체의 getNowPlaying() 메소드 실행 결과값에 대해 Ternary Operator를 적용한 테스트 메소드이다.

```
public void evaluateTernaryOperatorExpression() throws Exception {
    Movie movie = getGenre().getMovies().get(0);
    StandardEvaluationContext context = new StandardEvaluationContext(movie);

    ExpressionParser parser = new SpelExpressionParser();
    String playing = parser.parseExpression(
        "getNowPlaying().equals('Y') ? 'playing' : 'not playing'")
        .getValue(context, String.class);
    System.out
        .println("[Ternary Operator Expression] Movie 'Shrek (2001)' is " + playing);
}
```

위 메소드에서 활용한 테스트 데이터 Movie가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 Movie 객체의 getNowPlaying()의 값이 'N'이므로 'not playing'이라는 문자열이 리턴될 것이다. 따라서 위 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Ternary Operator Expression] Movie 'Shrek (2001)' is not playing
```

6.3.13.Elvis Operator

일반적으로 Ternary Operator를 활용하는 경우 다음과 같이 변수가 반복해서 표현되는 경우가 있다.

```
ticketPrice!=null ? ticketPrice : '8000'
```

Ternary Operator를 간략화한 형태인 Elvis Operator를 활용하면 Expression 정의가 단순해진다. (Groovy Language에서 사용되었으며 Elvis의 머리 모양과 닮았다 하여 Elvis Operator란 용어로 불리운다.)

```
ticketPrice ? : '8000'
```

다음은 Movie 객체의 getTicketPrice() 메소드 실행 결과값에 대해 Elvis Operator를 적용한 테스트 메소드이다.

```
public void evaluateElvisOperatorExpression() throws Exception {
```

```

Movie movie = getGenre().getMovies().get(1);
StandardEvaluationContext context = new StandardEvaluationContext(movie);

ExpressionParser parser = new SpelExpressionParser();
int ticketPrice = parser.parseExpression("getTicketPrice()?:'8000'")
    .getValue(context, Integer.class);
System.out
    .println("[Elvis Operator Expression] The ticket-price of 'Shrek (2001)' is "
        + ticketPrice);
}

```

위 메소드에서 활용한 테스트 데이터 Movie가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 Movie 객체의 getTicketPrice()의 값이 셋팅되어 있지 않으므로 즉, Null 값을 가지므로 Elvis Operator 실행 결과 '8000'이라는 int 타입의 값이 리턴될 것이다. 따라서 위 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Elvis Operator Expression] The ticket-price of 'Shrek (2001)' is 8000
```

6.3.14.Safe Navigation Operator

Groovy Language에서 사용된 Safe Navigation Operator는 Null 값을 가지는 객체를 대상으로 특정 메소드를 호출하거나 속성 정보에 접근하였을 경우 발생하는 NullPointerException을 방지하기 위해 활용 가능하다. 다음은 Movie 객체 내의 Genre가 Null 값을 가질 경우 Safe Navigation Operator를 활용하여 NullPointerException이 throw되는 것을 방지한 테스트 메소드이다.

```

public void evaluateSafeNavigationOperatorExpression() throws Exception {
    Movie movie = getGenre().getMovies().get(0);
    StandardEvaluationContext context = new StandardEvaluationContext(movie);

    ExpressionParser parser = new SpelExpressionParser();
    String name = parser.parseExpression("genre?.Name").getValue(context,
        String.class);
    System.out
        .println("[Save Navigation Operator Expression] The genre of 'Shrek (2001)' is "
            + name);

    movie.setGenre(null);
    name = parser.parseExpression("genre?.Name").getValue(context,
        String.class);
    System.out
        .println("[Safe Navigation Operator Expression] The genre of 'Shrek (2001)' is "
            + name);
}

```

위 메소드에서 활용한 테스트 데이터 Movie가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 첫번째 Expression ("genre?.Name")의 실행 결과는 'Adventure'가 될 것이다. 그리고 위 코드 중간에서 보는 바와 같이 Movie 객체 내의 Genre의 값을 Null로 변경한 후, 두번째 Expression ("genre?.Name")을 실행하였을 때에는 NullPointerException이 발생하지 않고 null 값이 리턴되어 실행 결과가 다음과 같이 출력될 것이다.

```
[Save Navigation Operator Expression] The genre of 'Shrek (2001)' is Adventure
[Safe Navigation Operator Expression] The genre of 'Shrek (2001)' is null
```

6.3.15.Collection Selection

?[selectionExpression] 형태의 Expression을 활용하면 대상이 되는 Collection 객체로부터 정의된 조건에 해당하는 Sub Collection을 도출할 수 있다. 또한 ?^[selectionExpression]는 정의된 조건에 부합하는

첫번째 데이터를, ?\$[selectionExpression]는 정의된 조건에 부합하는 마지막 데이터를 도출할 때 활용 가능하다. 다음은 Genre 객체 내의 java.util.ArrayList 유형의 movies로부터 Runtime 속성값이 90을 초과하는 Movie 정보를 추출하는 Collection Selection Expression을 포함하는 테스트 로직이다.

```
public void evaluateCollectionSelectionExpression() throws Exception {
    Genre genre = getGenre();
    StandardEvaluationContext context = new StandardEvaluationContext(genre);

    ExpressionParser parser = new SpelExpressionParser();
    List<Movie> movies = (List<Movie>) parser.parseExpression(
        "movies.?[Runtime > 90]").getValue(context);

    System.out.println("[Collection Selection Expression] Movie title is "
        + movies.get(0).getTitle());
}
```

위 메소드에서 활용한 테스트 데이터 Genre가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 Runtime 속성값이 '90'을 초과하는 Movie 객체는 1개이므로 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Collection Selection Expression] Movie title is Avatar
```

6.3.16.Collection Projection

![projectionExpression] 형태의 Expression을 활용하면 대상이 되는 Collection 객체로부터 Expression에 해당하는 정보만을 추출하여 생성한 새로운 Collection을 전달받을 수 있다. 다음은 Genre 객체 내의 java.util.ArrayList 유형의 movies로부터 'title' 정보만을 추출하는 Collection Projection Expression을 포함하는 테스트 메소드이다.

```
public void evaluateCollectionProjectionExpression() throws Exception {
    Genre genre = getGenre();
    StandardEvaluationContext context = new StandardEvaluationContext(genre);

    ExpressionParser parser = new SpelExpressionParser();
    List<String> titleList = (List<String>) parser.parseExpression(
        "movies.![title]").getValue(context);

    System.out
        .println("[Collection Projection Expression] first movie title is "
            + titleList.get(0) + ", second movie title is " + titleList.get(1));
}
```

위 메소드에서 활용한 테스트 데이터 Genre가 [테스트 데이터 : Genre & Movies]의 형태로 구성되어 있을 경우 'title' 값만을 추출하여 전달받은 새로운 'titleList'는 2개의 Title을 가지고 있을 것이며 메소드 실행 결과는 다음과 같이 출력될 것이다.

```
[Collection Projection Expression] first movie title is Shrek (2001), second movie title is Avatar
```

6.3.17.Expression Templating

Expression Template은 다수의 Evaluation Block과 정의된 문자열을 혼합할 수 있도록 지원한다. 각 Evaluation Block은 별도로 정의한 prefix, suffix로 구분지어 정의할 수 있다. Expression Template을 활용하기 위해서는 ExpressionParser.parseExpression() 메소드 호출시 첫번째 인자에는 Spring Expression, 두번째 인자에는 Expression Templating을 위한 ParserContext를 지정해 주어야 한다. Spring Framework에서는 기본적으로 활용할 수 있는 ParserContext로

org.springframework.expression.common.TemplateParserContext를 제공하며, TemplateParserContext는 '#{}'를 prefix, '}'를 suffix 구분자로 지정하고 있음을 알 수 있다.

```
public class TemplateParserContext implements ParserContext {
    public String getExpressionPrefix() { return "#{"; }
    public String getExpressionSuffix() { return "}"; }
    public boolean isTemplate() { return true; }
}
```

따라서 다음과 같이 Expression이 정의된 경우 randomPhrase의 값은 'random number is ####'와 같은 형태가 될 것이다.

```
ExpressionParser parser = new SpelExpressionParser();
String randomPhrase =
    parser.parseExpression("random number is #{T(java.lang.Math).random()}",
        new TemplateParserContext()).getValue(String.class);
```

6.3.18.테스트 데이터 : Genre & Movies

SpEL에서 지원하는 다양한 Expression 표현을 위해 활용된 Genre 객체는 getGenre() 메소드를 호출함으로써 얻어낼 수 있으며 Genre 객체는 다음과 같은 데이터를 포함하고 있다. 기본적으로 Genre는 기본 속성 정보 외에 java.util.ArrayList 유형의 Movie 목록을 포함하고 있다. 또한 Movie 객체는 기본 속성 정보 외에 관련된 Genre 객체를 포함하고 있다.

```
private Genre getGenre() {
    Genre genre = new Genre();
    genre.setGenreId("GR-02");
    genre.setName("Adventure");

    ArrayList<Movie> movies = new ArrayList<Movie>();

    Movie movie = new Movie();
    movie.setGenre(genre);
    movie.setMovieId("MV-000001");
    movie.setTitle("Shrek (2001)");
    movie.setActors("Shrek");
    movie.setDirector("Andrew Adamson");
    movie.setReleaseDate(new Date());
    movie.setRuntime(90);
    movie.setTicketPrice(8000);
    movie.setNowPlaying("N");
    movies.add(movie);

    movie = new Movie();
    movie.setGenre(genre);
    movie.setMovieId("MV-000002");
    movie.setTitle("Avatar");
    movie.setActors("Sigourney Weaver");
    movie.setDirector("James Cameron");
    movie.setReleaseDate(new Date());
    movie.setRuntime(100);
    movie.setNowPlaying("Y");
    movies.add(movie);

    genre.setMovies(movies);

    return genre;
}
```

6.4.Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, `mvn compile exec:java -Dexec.mainClass=...`이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, `src/main/java` 폴더 하위의 `Main.java`를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 `Run As > Java Application`을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
anyframe-sample-spel.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-spel.zip]

7.DataSource

주어진 Database에 연결하기 위한 Connection(javax.sql.Connection) 객체를 생성하는 서비스이다. Anyframe에서는 Connection Provider별로 Connection 객체를 얻어내기 위한 로직을 구현하고 있는 다음의 DataSource 구현체들을 그대로 사용하고자 한다.

7.1.JDBCDataSource Configuration

Description copied from class: SimpleDriverDataSource [http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/jdbc/datasource/SimpleDriverDataSource.html]

JDBC driver를 이용하여 Database Connection을 생성한다. 모든 getConnection() call에 대해 새로운 connection을 리턴한다. 실제 운영 환경에서는 JDBCDataSource의 사용은 추천하지 않으며, DBCPDataSource 나, C3P0DataSource 가 사용된다.

Property Name	Description	Required	Default Value
url	DataBase에 access하기 위한 JDBC URL	Y	N/A
driverClass	JDBC driver class name을 설정한다.	Y	N/A
username	DataBase에 access하기 위해 사용된다.	N	N/A
password	DataBase에 access하기 위해 사용된다.	N	N/A

7.1.1.Samples

다음은 JDBCDataSource의 속성 설정에 대한 예제이다.

- Configuration

다음은 JDBCDataSource의 속성을 정의한 context-datasource.xml의 일부이다. 아래 속성 정의 파일에서는 HSQL DB를 기반으로 한 JDBCDataSource Bean을 정의하고 있다.

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
  <property name="driverClass" value="org.hsqldb.jdbcDriver" />
  <property name="url" value="jdbc:hsqldb:file:./db/sampledb" />
  <property name="username" value="sa" />
</bean>
```

7.2.DBCPDataSource Configuration

JDBC driver를 이용하여 Database Connection을 생성하는 또다른 구현체이다.Commons DBCP [http://commons.apache.org/dbcp/] 라 불리는 Jakarta의 Database Connection Pool이다. Configuration parameter 전체 DBCP documentation [http://commons.apache.org/dbcp/configuration.html]을 통해 확인 가능하다.

Property Name	Description	Required	Default Value
driverClassName	jdbc driver의 class name을 설정한다.	Y	N/A
url	DataBase url을 설정한다.	Y	N/A
username	DataBase에 접근시 사용할 username을 설정한다.	N	N/A
password	DataBase에 접근시 사용할 password를 설정한다.	N	N/A

Property Name	Description	Required	Default Value
maxActive	동시에 할당할 수 있는 active connection의 최대 갯수를 설정한다.	N	8
maxIdle	pool에 남겨놓을 수 있는 idle connection의 최대 갯수를 설정한다.	N	8
maxWait	모든 Connection이 사용중일 경우 최대 대기 시간을 설정한다.	N	indefinitely
defaultAutoCommit	이 datasource로부터 리턴된 connection에 대한 auto-commit 여부를 설정한다.	N	true
defaultReadOnly	Connection Pool에 의해 생성된 Connection에 read-only 속성을 부여한다.	N	driver default
defaultTransactionIsolation	리턴된 connection에 대한 transaction isolation 속성을 부여한다.	N	driver default
defaultCatalog	Connection의 catalog를 설정한다.	N	N/A
minIdle	Connection pool의 최소한 idle connection 갯수를 설정한다.	N	0
initialSize	Connection pool에 생성될 초기 connection size를 설정한다.	N	0
testOnBorrow	Connection pool에서 객체를 가지고 오기 전에 그 객체의 유효성을 확인할 것인지 결정한다. true값은 아무 영향을 미치지 않지만 validationQuery property는 non-null string으로 설정되어야 한다.	N	true
testOnReturn	객체를 return하기 전에 객체의 유효성을 확인할 것인지 결정한다. true값은 아무 영향을 미치지 않지만 validationQuery property는 non-null string으로 설정되어야 한다.	N	false
testWhileIdle	idle object evictor가 connection의 유효성을 확인할 것인지를 설정한다. true값은 아무 영향을 미치지 않지만 validationQuery property는 non-null string으로 설정되어야 한다.	N	false
validationQuery	validationQuery를 설정한다.	N	N/A
loginTimeout	Database에 연결하기 위한 login timeout(in seconds)을 설정한다. createDataSource()를 호출 해서 connection pool을 초기화한다.	N	N/A

7.2.1.Samples

다음은 DBCPDataSource의 속성 설정에 대한 예제이다.

- **Configuration**

다음은 DBCPDataSource의 속성을 정의한 context-datasource.xml 의 일부이다. 아래 속성 정의 파일에서는 HSQL DB를 기반으로 한 DBCPDataSource Bean을 정의하고 있다.

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:file:./db/sampledb"/>
    <property name="username" value="sa"/>
    <property name="maxActive" value="100"/>
</bean>
```

```

<property name="maxIdle" value="30"/>
<property name="maxWait" value="1000"/>
<property name="defaultAutoCommit" value="true"/>
<property name="removeAbandoned" value="true"/>
<property name="removeAbandonedTimeout" value="60"/>
<property name="logAbandoned" value="true"/>
</bean>

```

- **Test case**

예제 코드는 Test Case 에 포함되어 있다.

7.3.C3P0DataSource Configuration

JDBC driver 를 이용하여 Database Connection을 생성하는 또다른 구현체이다. C3P0 Library에 관한 자세한 사항은 C3P0 Configuration [<http://www.mchange.com/projects/c3p0/index.html#c3p0-config.xml>]에서 확인할 수 있다.

7.3.1.Samples

다음은 C3P0DataSource의 속성 설정에 대한 예제이다.

- **Configuration**

다음은 C3P0DataSource의 속성을 정의한 context-datasource.xml의 일부이다. 아래 속성 정의 파일에서는 HSQL DB를 기반으로 하는 C3P0DataSource Bean을 정의하고 있다.

```

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
    destroy-method="close">
    <property name="driverClass" value="org.hsqldb.jdbcDriver"/>
    <property name="jdbcUrl" value="jdbc:hsqldb:file:./db/sampledb"/>
    <property name="user" value="sa"/>
    <property name="minPoolSize" value="5"/>
    <property name="acquireIncrement" value="5"/>
    <property name="maxPoolSize" value="15"/>
</bean>

```

7.4.JNDIDataSource Configuration

JNDIDataSource는 JNDI Lookup을 이용하여 Database Connection을 생성한다. JNDIDataSource는 대부분 Enterprise application server에서 제공되는 JNDI tree로 부터 DataSource를 가져온다.

Description copied from class: JndiObjectFactoryBean [<http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/jndi/JndiObjectFactoryBean.html>]

JNDIDataSource는 일반적으로 application context의 singleton factory(e.g.JNDI-bound DataSource)를 등록하여 사용할 수 있고, 필요한 application service를 빈으로 참조할 수 있다.

기본적으로 startup시 캐싱된 JNDI 객체를 검색한다. 이것은 "lookupOnStartup"과 "cache" property를 통해 customized 할 수 있으며, JndiObjectTargetSource를 사용할 수 있다. 실제 JNDI object type이 미리 정의되어 있지 않은 경우 proxyInterface의 정의가 필요하다.

Property Name	Description	Required	Default Value
jndiTemplate	JNDI 검색을 위해 JNDI 템플릿을 설정한다. 또한 "jndiEnvironment"로 JNDI 환경설정을 할 수 있다.	N	N/A
jndiEnvironment	JNDI를 검색하기 위해 JNDI 환경을 설정한다. 환경 설정에 제공된 JndiTemplate을 생성한다.	N	N/A

Property Name	Description	Required	Default Value
resourceRef	JavaEE 컨테이너에서 검색할 수 있는지 설정한다. 만약 prefix가 "java:comp/env/"이면 JNDI 이름이 포함되어 있지 않으므로 추가해 주어야 한다. 디폴트 값은 "false"이다. 주의 : 만약 "java:" 와 같이 주어진 scheme이 아니라면 적용할 수 없다.	N	false
expectedType	JNDI 객체의 타입을 지정한다.	N	N/A
jndiName	검색을 위해 JNDI 이름을 설정한다. 만약 resourceRef가 true로 설정되어 있고, "java:comp/env/"로 시작되지 않으면 이 prefix를 추가한다.	Y	N/A
proxyInterface	검색을 위해 JNDI 이름을 설정한다. 만약 resourceRef가 true로 설정되어 있고, "java:comp/env/"로 시작되지 않으면 이 prefix를 추가한다.	N	N/A
lookupOnStartup	startup시에 JNDI object를 검색할 지 여부를 설정한다. lazy lookup시에는 proxy interface 정의가 필요하다.	N	true
cache	JNDI 객체를 캐싱할 것인지 설정한다.	N	true
defaultObject	JNDI lookup에 실패하였을 경우 전달할 default object를 지정한다. 이것은 임의의 bean reference 나 literal value가 될 수 있다. 주의 : 이것은 startup 시 lookup에서만 지원된다.	N	none

7.4.1.Samples

다음은 JNDIDataSource의 속성 설정에 대한 예제이다. "jnditemplate" Bean에 JNDI Server에 대한 속성을 정의하고, "dataSource" Bean에서 "jnditemplate" Bean을 참조하여 Connection 객체를 얻어낼 수 있도록 하고 있다.

- **Configuration**

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="AnyframedS"/>
    <property name="jndiTemplate" ref="jnditemplate"/>
</bean>

<bean id="jnditemplate" class="org.springframework.jndi.JndiTemplate">
    <property name="environment">
        <props>
            <prop key="java.naming.factory.initial">
                weblogic.jndi.WLInitialContextFactory
            </prop>
            <prop key="java.naming.provider.url">
                t3://server.ip:7001
            </prop>
        </props>
    </property>
</bean>
```

7.4.2.jee schema 를 통한 JNDIDataSource 사용

Spring 2.0 이후 버전에서는 jee Namespace 태그를 통해 JNDI 객체를 lookup 할 수 있는 간소한 설정을 지원한다. 아래에서는 jee:jndi-lookup 를 사용한 JNDIDataSource 설정이다.

- **Configuration**

```
<?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:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-4.0.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="AnyframeDS" resource-ref="true">
    <jee:environment>
      java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
      java.naming.provider.url=t3://server.ip:7001
    </jee:environment>
  </jee:jndi-lookup>
</beans>
```

JndiObjectFactoryBean 와 JndiTemplate 을 통한 설정에 비해 jee 태그를 사용하면 설정이 매우 간소하므로 이 방법을 사용할 것을 권고한다. jee schema 에 대한 상세 내용은 이곳 [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/xsd-config.html]을 참고하도록 한다.

7.5. Test Case

다음은 앞서 정의한 속성 설정 파일들을 기반으로 하여 DataSource로부터 connection을 가져오는 Main.java 코드의 일부이다.

```
public void getConnection() throws Exception {
    // 1. lookup dataSource
    DataSource datasource = (DataSource) context.getBean("dataSource");
    // 2. try to get a connection from dbcp connection pool
    Connection conn = datasource.getConnection();
    System.out.println("Connection is " + conn + "");
}
```

7.6. Resources

- 다운로드

다음에서 테스트 DB를 포함하고 있는 hsqldb.zip과 sample 코드를 포함하고 있는 anyframe-sample-datasource.zip 파일을 다운받은 후, 압축을 해제한다. 그리고 hsqldb 폴더 내의 start.cmd (or start.sh) 파일을 실행시켜 테스트 DB를 시작시켜 놓는다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, mvn compile exec:java -Dexec.mainClass=...이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, src/main/java 폴더의 anyframe/sample/datasource 하위의 Main.java를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 Run As > Java Application을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
hsqldb.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/hsqldb.zip]
anyframe-sample-datasource.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-datasource.zip]

- 참고자료

- JDBCDataSource - SimpleDriverDataSource [<http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/jdbc/datasource/SimpleDriverDataSource.html>]
- DBCP Configuration [<http://commons.apache.org/dbcp/configuration.html>]
- C3P0 Configuration [<http://www.mchange.com/projects/c3p0/index.html#c3p0-config.xml>]
- JNDIDataSource - JndiObjectFactoryBean [<http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/jndi/JndiObjectFactoryBean.html>]
- JNDIDataSource - jee schema [<http://static.springsource.org/spring/docs/4.0.x/spring-framework-reference/html/xsd-config.html>]

8.Transaction Management

Transaction 관리에 대하여 일관성 있는 추상화된 방법을 제공하는 서비스로 다음과 같은 장점을 제공한다.

- JTA, JDBC와 같은 서로 다른 Transaction API에 대해 일관성 있는 프로그래밍 모델을 제공한다.
- 프로그램적인 Transaction 관리에 대한 사용하기 쉬운 API를 제공한다.
- 선언적인 Transaction 관리를 지원한다.
- Hibernate와 통합이 용이하다.

다음 목록에 제시된 Transaction 서비스 중, 적합한 서비스를 선택하여 Transaction을 관리할 수 있다.

DataSource Transaction Service

DataSource Transaction 서비스는 DataSource를 사용하여 Local Transaction을 관리한다.

• Samples

다음은 DataSource Transaction 서비스의 속성 설정 및 테스트 코드에 대한 예제이다.

• Configuration

다음은 DataSourceTransactionManager의 속성 정의 파일인 context-transaction-datasource.xml의 일부이다. 여기에서는 dataSource property를 정의해 주어야 한다.

```
<bean id="transactionManagerDataSource"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource"><ref bean="common_datasource"/></property>
</bean>
```

위에서 제시한 common_datasource Bean은 DataSource 서비스의 속성을 정의한 context-datasource-common.xml 파일에 다음과 같이 정의되어 있다.

```
<bean id="common_datasource"
      class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:file:./db/sampledb"/>
  <property name="username" value="sa"/>
</bean>
```

• TestCase

다음은 앞서 정의한 속성 설정을 기반으로 하여 DataSource Transaction 서비스를 이용하여 Transaction 처리 기능을 테스트 하는 TransactionServiceTestDataSource.java 코드의 일부이다. 실제 테스트 메소드는 AbstractTransactionServiceTest.java 에 공통으로 작성되어 있다. DataSource Transaction 서비스를 테스트하기 위해서는 setUp() 메소드를 다음과 같이 작성해준다.

```
public class TransactionServiceTestDataSource extends AbstractTransactionServiceTest {
  중략...
  protected void setup() {
    super.setup();
    this.service = (TransactionTestSampleService)context
      .getBean("transactionSampleDataSource");
    this.transactionManager = (PlatformTransactionManager)context
      .getBean("transactionManagerDataSource");
  }
}
```

```
}
```

Hibernate Transaction Service

Hibernate Transaction 서비스는 DataSource를 사용하여 Local Transaction과 Hibernate Session을 관리한다. HibernateTransactionManager는 SessionFactoryBean에 의존성을 가지고 있으므로 반드시 SessionFactoryBean 설정과 함께 정의되어야 한다.

- **Samples**

다음은 Hibernate Transaction 서비스의 속성 설정 및 테스트 코드에 대한 예제이다.

- **Configuration**

다음은 HibernateTransactionManager의 속성 정의 파일인 context-transaction-hibernate.xml의 일부이다. 여기에서 sessionFactory property를 정의해 주는데, sessionFactory 설정은 sessionFactory Bean을 참조하고 있음을 알 수 있다. Hibernate의 sessionFactory Bean 설정 방법은 Hibernate [http://dev.anyframejava.org/docs/anyframe/plugin/optional/hibernate/1.6.0/reference/htmlsingle/hibernate.html#hibernate_hibernate_springintegration_property] 서비스 매뉴얼 내용을 참고한다.

```
<bean id="transactionManagerHibernate"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

위에서 sessionFactory Bean은 Hibernate 서비스 속성을 정의한 context-hibernate.xml 파일에 다음과 같이 정의되어 있다.

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="common_datasource" />
    <property name="mappingDirectoryLocations">
        <value>classpath:/spring/services/hibernate</value>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</prop>
            <prop key="hibernate.cache.use_second_level_cache">true</prop>
        </props>
    </property>
</bean>
```

- **TestCase**

다음은 앞서 정의한 속성 설정을 기반으로 하여 Hibernate Transaction 서비스를 이용하여 Transaction 처리 기능을 테스트 하는 TransactionServiceTestHibernate.java 코드의 일부이다. 실제 테스트 메소드는 AbstractTransactionServiceTest.java 에 공통으로 작성되어 있다. Hibernate Transaction 서비스를 테스트하기 위해서는 setUp() 메소드를 다음과 같이 작성해준다.

```
public class TransactionServiceTestHibernate extends AbstractTransactionServiceTest {

    중략...
    protected void setup() {
        super.setup();
        this.service =
```

```

        (TransactionTestSampleService)context.getBean("transactionSampleHibernate");
        this.transactionManager =
            (PlatformTransactionManager)context.getBean("transactionManagerHibernate");
    }
}

```

JTA Transaction Service

JTA Transaction 서비스는 JTA를 사용하여 Global Transaction 관리 부분을 추상화하고, 해당 서비스로 인해 JTA, JNDI 등에 종속되지 않게 구현할 수 있도록 도와준다. 또한 이때 DataSource 서비스는 JNDI DataSource 서비스로 설정해줘야 한다.

• Samples

다음은 JTA Transaction 서비스의 속성 설정 및 테스트 코드에 대한 예제이다.

• Configuration

다음은 JTATransactionManager의 속성 정의 파일인 context-transaction-weblogic.xml의 일부이다. 여기에서 transactionManagerName property와 jnditemplate property를 정의해 주도록 한다. transactionManagerName property는 해당 WAS에 등록된 TransactionManager의 JNDI명을 정의해 주어야 하는데, 이것은 WAS 벤더 별로 다를 수 있음에 유의하도록 한다. 또한 jnditemplate property에는 해당하는 WAS의 provider url과 initial context factory 클래스명을 정의해주면 된다.

```

<bean id="transactionManagerWebLogic"
    class="org.springframework.transaction.jta.WebLogicJtaTransactionManager">
    <property name="transactionManagerName"
        value="javax.transaction.TransactionManager" />
    <property name="jndiTemplate" ref="jndiTemplate"></property>
</bean>

<bean id="jndiTemplate" class="org.springframework.jndi.JndiTemplate" >
    <property name="environment">
        <props>
            <prop key=
                "java.naming.factory.initial">weblogic.jndi.WLInitialContextFactory</prop>
            <prop key="java.naming.provider.url">t3://server.ip:7001</prop>
        </props>
    </property>
</bean>

```

또한, JEUS Server를 통해 Transaction을 관리하고자 하는 경우에는 다음과 같이 TransactionManager를 설정할 수 있다.

```

<bean id="transactionManagerJEUS"
    class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManagerName" value="java:/TransactionManager" />
    <property name="jndiTemplate" ref="jndiTemplate"></property>
</bean>

<bean id="jndiTemplate" class="org.springframework.jndi.JndiTemplate" >
    <property name="environment">
        <props>
            <prop key="java.naming.factory.initial">jeus.jndi.JNSContextFactory</prop>
            <prop key="java.naming.provider.url">server.ip:9736</prop>
        </props>
    </property>
</bean>

```

JEUS Server는 추가 설정없이도 기본적으로 TransactionManager를 가지고 있으며 해당 TransactionManager의 JNDI명은 java:/TransactionManager 이다. JNDI 명이 java:/

TransactionManager인 TransactionManager를 찾지 못해 javax.naming.NameNotFoundException이 발생하였다면 해당 프로젝트의 클래스패스 내에 j2ee-x.x.jar 파일이 추가되어 있지 않은지 확인하여 삭제하도록 한다. 이는 JEUS Server를 위한 기본 라이브러리 내에 존재하는 Transaction 처리 관련 클래스와의 충돌로 인해 발생하는 문제이기 때문이다.

Spring 2.5 에서 새롭게 추가된 tx:jta-transaction-manager 설정 요소를 사용하면 JTA 기반 트랜잭션 플랫폼을 자동 탐지하여 적절한 PlatformTransactionManager를 등록해 주게 된다. 이때 transaction manager bean id 는 "transactionManager"로 자동 설정됨에 유의한다. 다음은 JTATransactionManager의 속성 정의 파일인 context-transaction-jta.xml의 일부이다. 여기에서 JTATransactionManager는 JNDIDataSource와 함께 사용되므로 아래에서 같은 설정 파일내에 표시하였으나, 다른 설정 파일(context-datasource.xml)에 설정하는 경우가 대부분이다.

```
<?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:jee="http://www.springframework.org/schema/jee"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-4.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

    <tx:jta-transaction-manager/>

    <jee:jndi-lookup id="dataSource" jndi-name="AnyFrameDS" resource-ref="true">
        <jee:environment>
            java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
            java.naming.provider.url=t3://server.ip:7001
        </jee:environment>
    </jee:jndi-lookup>

</beans>
```

IBM WebSphere, BEA Weblogic, Oracle OC4J 에 대해서는 벤더 Specific한(버전 유의) 어댑터 클래스를 사용할 수도 있다.

• TestCase

다음은 앞서 정의한 속성 설정을 기반으로 하여 JTA Transaction 서비스를 이용하여 Transaction 처리 기능을 테스트 하는 TransactionServiceTestJTA.java 코드의 일부이다. 실제 테스트 메소드는 AbstractTransactionServiceTest.java에 공통으로 작성되어 있다. DataSource Transaction 서비스를 테스트하기 위해서는 setUp() 메소드를 다음과 같이 작성해준다.

```
public class TransactionServiceTestHibernate extends AbstractTransactionServiceTest {
    중략...
    protected void setup() {
        super.setup();
        this.service = (TransactionTestSampleService)context
            .getBean("transactionManager");
        this.transactionManager = (PlatformTransactionManager)context
            .getBean("transactionManager");
    }
}
```

예제 테스트 코드는 런타임시 WebLogic 라이브러리를 참조하므로 [WebLogic Home]/server/lib/폴더의 weblogic-9.2.jar, xbean-9.2.jar 파일을 참조 라이브러리로 적절히 설정해야 한다. 위 작업이 완료된 후, WebLogic Server가 성공적으로 시작된 상태에서 예제 테스트 코드를 실행하도록 한다.

8.1. Declarative Transaction Management

본 문서에서는 코드에서 직접적으로 Transaction 처리하지 않고, 선언적으로 Transaction을 관리할 수 있는 방법에 대해 살펴보기로 하자. Spring에서는 선언적인 Transaction 관리를 위해 다양한 방법을 제공한다.

8.1.1. Annotation을 이용한 Transaction 관리

8.1.1.1. Configuration

본 매뉴얼 >> Spring >> Transaction Management를 참고하여 Transaction 서비스의 속성을 정의한다. 다음은 Transaction 서비스의 속성을 정의한 XML(context-tx.xml) 파일로, Transaction을 관리하는 실질적 역할을 수행하는 TransactionManager가 정의되어 있다.

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
```

또한, @Transactional이 적용된 클래스들을 런타임시에 Proxy 클래스로 대체시켜, Transaction 관리 대상인 메소드가 호출되면 Proxy에서 Transaction 서비스를 통해 Transaction이 시작된 후 해당 메소드가 호출될 수 있게 하기 위해 Spring 속성 정의 XML 파일에 다음과 같이 추가해주어야 한다.

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

8.1.1.2. Transaction 관리 대상 정의

Spring에서 제공하는 @Transactional Annotation을 이용하여 Transaction 관리 대상 클래스 또는 메소드를 식별한다. 다음은 MovieServiceImplWithAnnotation 의 일부로 전체 클래스에 대해 Transaction 관리 여부를 정의하고 있음을 알 수 있다. 또한 @Transactional Annotation은 메서드위에 나타낼 수도 있으며 상제 속성을 지시할 수도 있다. 단, 다수의 Transaction Manager가 정의되어 있고 Annotation 기반에서 Transaction을 관리하고자 하는 경우 특정 클래스에 대해 Transaction Manager를 지정하여 활용할 수 없음에 유의해야 한다. (Spring 2.5 이하)

```
@Service("annotationMovieService")
@Transactional
public class MovieServiceImplWithAnnotation implements MovieService {

    .. 중략

    @Transactional(noRollbackFor = { MovieException.class }, propagation =
        Propagation.REQUIRED)
    public void updateMovieList(Movie newMovie, Movie updateMovie)
        throws MovieException {
        String movieName = "";
        try {
            movieName = newMovie.getTitle();
            create(newMovie);

            movieName = updateMovie.getTitle();
            int result = update(updateMovie);
            if (result <= 0) {
                throw new MovieException("fail to update with wrong movieid.");
            }
        }
    }
}
```



```

    } catch (Exception e) {
        throw new MovieException("'" + movieName
            + "' - Failed to update movie data");
    }
}
}

```

위 샘플 코드에 정의된 바와 같이 Transaction 관리를 위해 @Transactional Annotation에는 다음과 같은 상세 속성 정보를 부여할 수 있다.

속성	설명
isolation	Transaction의 isolation Level 정의하는 요소. 별도로 정의하지 않으면 DB의 Isolation Level을 따름. Isolation.DEFAULT, Isolation.READ_COMMITTED, Isolation.READ_UNCOMMITTED, Isolation.REPEATABLE_READ, Isolation.SERIALIZABLE 중 선택하여 정의할 수 있다. (Default = Isolation.DEFAULT) 각 Isolation Level에 대한 자세한 내용은 본 페이지의 [참고] Propagation Behavior, Isolation Level 를 참고하도록 한다.
noRollbackFor	정의된 Exception 목록에 대해서는 rollback을 수행하지 않음.
noRollbackForClassname	Class 객체가 아닌 문자열을 이용하여 rollback을 수행하지 않아야 할 Exception 목록 정의
propagation	Transaction의 propagation 유형을 정의하기 위한 요소. Propagation.MANDATORY, Propagation.NESTED, Propagation.NEVER, Propagation.NOT_SUPPORTED, Propagation.REQUIRED, Propagation.REQUIRES_NEW, Propagation.SUPPORTS 중 선택하여 정의할 수 있다. (Default = Propagation.REQUIRED) 각 Propagation 유형에 대한 자세한 내용은 본 페이지의 [참고] Propagation Behavior, Isolation Level 를 참고하도록 한다.
readOnly	해당 Transaction을 읽기 전용 모드로 처리 (Default = false)
rollbackFor	정의된 Exception 목록에 대해서는 rollback 수행
rollbackForClassName	Class 객체가 아닌 문자열을 이용하여 rollback을 수행해야 할 Exception 목록 정의
timeout	지정한 시간 내에 해당 메소드 수행이 완료되지 않은 경우 rollback 수행. -1일 경우 no timeout (Default = -1)

8.1.1.3.테스트 클래스 실행

MovieServiceImplWithAnnotation 클래스 레벨에 정의된 Annotation을 이용한 Transaction 관리 방법에 대한 테스트는 기본 샘플 실행 코드인 org.anyframe.sample.transaction.Main.java 를 참조하도록 한다.

8.1.2.XML 정의를 이용한 Transaction 관리

8.1.2.1.Configuration

본 매뉴얼 >> Spring >> Transaction Management 을 참고하여 Transaction 서비스의 속성을 정의한다. 다음은 Transaction 서비스의 속성을 정의한 XML(context-tx.xml) 파일로, Transaction을 관리하는 실질적 역할을 수행하는 TransactionManager가 정의되어 있다.

```

<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

```

```
<property name="dataSource">
  <ref bean="dataSource"/></property>
</bean>
```

8.1.2.2.Transaction 관리 대상 정의

다음 Spring 속성 정의 XML(context-movie.xml.xml)과 같이 <tx:advice>와 <aop:config>를 이용하여 Advice와 Pointcut을 정의한다. (단, <tx:advice>와 <aop:config>를 이용하기 위해서는 tx, aop namespace에 대한 정의가 필요하다.)

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

  <tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
      <!-- 메소드 실행중 MovieException이 발생한 경우 rollback을 수행하지 않음 -->
      <tx:method name="*" no-rollback-for="org.anyframe.sample.exception.MovieException"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <!-- pointcut 정의 : UserServiceImplWithXML 클래스의 모든 메소드 호출시 -->
    <aop:pointcut id="movieServiceOperations"
      expression="
        *execution(*
          org.anyframe.sample.transaction.moviefinder.service.impl.MovieServiceImpl.*(..))"/>
    <!-- advice 정의 : 위 tx 태그를 이용하여 정의한 advice 참조 -->
    <aop:advisor advice-ref="txAdvice" pointcut-ref="movieServiceOperations"/>
  </aop:config>
```

위 샘플 XML에서와 같이 Transaction 관리를 위해 <tx:advice> 하위 태그인 <tx:method>에는 다음과 같은 상세 속성 정보를 부여할 수 있다.

속성	설명
name	메소드명. 와일드카드 사용 가능
isolation	Transaction의 isolation Level을 정의하는 요소. 별도로 정의하지 않으면 DB의 Isolation 레벨을 따름. DEFAULT, READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, SERIALIZABLE 중 선택하여 정의할 수 있다. (Default = DEFAULT) 각 Isolation Level에 대한 자세한 내용은 본 페이지의 [참고] Propagation Behavior, Isolation Level 를 참고하도록 한다.
no-rollback-for	정의된 Exception 목록에 대해서는 rollback을 수행하지 않음.
propagation	Transaction의 propagation 유형을 정의하기 위한 요소. MANDATORY, NESTED, NEVER, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW, SUPPORTS 중 선택하여 정의할 수 있다. (Default = REQUIRED) 각 Propagation 유형에 대한 자세한 내용은 본 페이지의 [참고] Propagation Behavior, Isolation Level 를 참고하도록 한다.
read-only	해당 Transaction을 읽기 전용 모드로 처리. (Default = false)

속성	설명
rollback-for	정의된 Exception 목록에 대해서는 rollback 수행
timeout	지정한 시간 내에 해당 메소드 수행이 완료되지 않은 경우 rollback 수행. -1일 경우 no timeout. (Default = -1)

8.1.2.3.테스트 클래스 실행

XML 정의를 이용한 Transaction 관리 방법에 대한 테스트는 context-movie.xml.xml 의 updateMovieList 메소드에 정의된 트랜잭션 관리 속성 정보를 기반으로 한다. 테스트 클래스 구성은 본 페이지 내의 테스트 케이스 상세 를 참조하며, 보다 자세한 코드는 DeclarativeTransactionManagementWithXMLTest 를 참조하도록 한다.

8.1.3.[참고] Propagation Behavior, Isolation Level

다음에서는 Transaction 속성값으로 정의할 수 있는 Propagation Behavior와 Isolation Level에 보다 자세히 알아보기로 하자.

8.1.3.1.Propagation Behavior

Propagation Behavior(전달 행위)는 Transaction 전파 규칙을 정의하기 위해 사용된다.

Attribute Name	Description
PROPAGATION_MADATORY	반드시 Transaction 내에서 메소드가 실행되어야 하고, Transaction이 없는 경우에는 예외를 발생시킨다.
PROPAGATION_NESTED	Transaction에 있는 경우, 기존 Transaction 내의 nested transaction 형태로 메소드를 실행하고, nested transaction 자체적으로 commit, rollback이 가능하다. Transaction이 없는 경우, PROPAGATION_REQUIRED 속성으로 행동한다.
PROPAGATION_NEVER	Transaction 컨텍스트 없이 실행되어야 하며 Transaction이 있으면 예외를 발생시킨다.
PROPAGATION_NOT_SUPPORTED	Transaction 없이 메소드를 실행하며, 기존의 Transaction이 있는 경우에는 이 Transaction을 호출된 메소드가 끝날 때까지 잠시 보류한다.
PROPAGATION_REQUIRED	Transaction 컨텍스트 내에서 메소드가 실행되어야 한다. 기존 Transaction이 있는 경우에는 기존 Transaction 내에서 실행하고, 기존 Transaction이 없는 경우에는 새로운 Transaction을 생성한다.
PROPAGATION_REQUIRED_NEW	호출되는 메소드는 자신 만의 Transaction을 가지고 실행하고, 기존의 Transaction들은 보류된다.
PROPAGATION_SUPPORTS	새로운 Transaction을 필요로 하지는 않지만, 기존의 Transaction이 있는 경우에는 Transaction 내에서 메소드를 실행한다.

8.1.3.2.Isolation Level

Isolation Level(격리수준)은 Transaction에서 일관성이 없는 데이터를 허용하도록 하는 수준이며, 여러 Transaction들이 다른 Transaction의 방해로부터 보호되는 정도를 나타낸다. 예를 들어, 한 사용자가 어떠한 데이터를 수정하고 있는 경우 다른 사용자들이 그 데이터에 접근하는 것을 차단함으로써 완전한 데이터만을 사용자들에게 제공하게 된다. 또한, 많은 사용자들의 수정 작업으로 인하여 통계 자료를 작성할 수 없는 사용자를 위하여 읽기 작업을 수행할 수 있도록 Isolation Level을 변경할 수 있다.

Attribute Name	Description
ISOLATION_DEFAULT	개별적인 PlatformTransactionManager를 위한 디폴트 격리 레벨
ISOLATION_READ_COMMITTED	이 격리수준을 사용하는 메소드는 commit 되지 않은 데이터를 읽을 수 없다. 쓰기 락은 다른 Transaction에 의해 이미 변경된 데이터는 얻을 수 없다. 따라서 조회 중인 commit 되지 않은 데이터는 불가능하다. 대개의 데이터베이스에서의 디폴트로 지원하는 격리 수준이다.
ISOLATION_READ_UNCOMMITTED	가장 낮은 Transaction 수준이다. 이 격리수준을 사용하는 메소드는 commit 되지 않은 데이터를 읽을 수 있다. 그러나 이 격리수준은 새로운 레코드가 추가되었는지 알 수 없다.
ISOLATION_REPEATABLE_READ	ISOLATION_READ_COMMITTED 보다는 다소 조금 더 엄격한 격리 수준이다. 이 격리 수준은 다른 Transaction이 새로운 데이터를 입력했다면, 새롭게 입력된 데이터를 조회할 수 있다는 것을 의미한다.
ISOLATION_SERIALIZABLE	가장 높은 격리수준이다. 모든 Transaction(조회를 포함하여)은 각 라인이 실행될 때마다 기다려야 하기 때문에 매우 느리다. 이 격리수준을 사용하는 메소드는 데이터 상에 배타적 쓰기 락을 얻음으로써 Transaction이 종료될 때까지 조회, 수정, 입력 데이터로부터 다른 Transaction의 처리를 막는다. 가장 많은 비용이 들지만 신뢰할만한 격리 수준을 제공하는 것이 가능하다.

8.1.4.테스트 케이스 상세

다음은 앞서 언급한 Annotation을 이용한 Transaction 관리, XML 정의를 이용한 Transaction 관리 방법을 테스트해 보기 위해 동일한 로직으로 구성된 테스트 클래스의 일부이다. 각 테스트 클래스의 testUpdateMovieWithNotExistMovie() 메소드에서는 MovieService의 updateMovieList 메소드를 호출한다. updateMovieList 메소드의 첫번째 입력 인자는 신규 무비로 두번째 입력 인자는 수정할 무비로 인식된다. 따라서 첫번째 입력 인자를 신규 무비 정보로 하고 두번째 입력 인자를 존재하지 않는 무비ID를 가진 무비 정보로 전달하였을 경우 신규 무비를 성공적으로 등록하고 두번째 특정 무비 정보를 수정하려고 했을 때 해당 ID의 무비가 존재하지 않아 수정에 실패하게 된다. MovieService의 updateMovieList 메소드에 대해 정의된 Transaction 속성에 의해 신규로 등록한 무비 정보가 rollback되지 않고 commit 되어야 한다.

```
public void testUpdateMovieWithNotExistMovie() throws Exception {
    Movie newMovie = getMovie();
    String newMovieID = newMovie.getMovieId();
    Movie updateMovie = movieService.get("MV-00003");
    // 존재하지 않는 무비 ID로 변경
    updateMovie.setMovieId("MV-11111");
    updateMovie.setTitle("TEST Movie");

    try {
        // 신규 무비 정보와 수정 대상 무비 정보를 인자로 셋팅
        movieService.updateMovieList(newMovie, updateMovie);
        // 해당 메소드 수행시 예외가 발생하지 않으면 fail
        fail("fail to get user.");
    } catch (Exception e) {
        try {
            // 신규 무비의 ID로 무비 정보 조회
            Movie movie = movieService.get(newMovieID);
            assertNotNull("fail to commit.", movie);
        } catch (Exception ie) {
            // 무비 정보 수정에 실패하였으나 Annotation 정의에 따라
        }
    }
}
```

```

        // 신규 등록된 무비 정보는 commit되었어야 함.
        fail("fail to transaction management.");
    }
}
}

```

8.2. Programmatic Transaction Management

본 문서에서는 세밀한 Transaction 제어가 필요한 경우 코드 내에서 직접적으로 Transaction을 처리하는 방법에 대해 살펴보기로 하자. Spring에서는 프로그램적 Transaction 관리를 위해 다음과 같이 2가지 방법을 제공한다.

8.2.1. TransactionTemplate을 이용한 Transaction 관리

8.2.1.1. Configuration

TransactionTemplate을 이용하여 Transaction을 관리하기 위해서는 Transaction 서비스와 TransactionTemplate에 대한 속성 정의가 필요하다. 다음은 TransactionTemplate에 대한 속성 정의 파일(context-transaction.xml)의 일부로 transactionManager property에 대한 정의를 필요로 한다.

```

<bean id="transactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="transactionManager"/>
</bean>

```

다음은 Transaction 서비스의 속성을 정의한 XML(context-transaction.xml) 파일로, Transaction을 관리하는 실질적 역할을 수행하는 TransactionManager가 정의되어 있다.

```

<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>

```

Transaction 서비스의 속성 정의시 본 매뉴얼 >> Spring >> Transaction Management 을 참고하도록 한다.

8.2.1.2. Transaction 관리

TransactionTemplate을 이용하여 프로그램적인 방법으로 Transaction을 관리하고자 하는 경우, Transaction Context에 의해 호출될 callback 메소드를 정의하고 이 메소드 내에 비즈니스 로직을 구현해 주면 된다.

```

this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {
    public void doInTransactionWithoutResult(TransactionStatus status) {
        //... biz. logic ...
    }
});

this.transactionTemplate.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {

```

```
//... biz. logic ...
});
```

callback 메소드 `doInTransactionWithoutResult()`는 전달할 값이 없는 경우에 정의 가능하며, 전달해야 하는 값이 존재하는 경우에는 `doInTransaction()`으로 정의하도록 한다. 또한, callback 메소드 내에서 입력 인자인 `TransactionStatus` 객체의 `setRollbackOnly()` 메소드를 호출함으로써 해당 Transaction을 rollback할 수 있다.

8.2.1.3. 테스트 클래스 실행

테스트 클래스 `MovieServiceWithProgrammaticTest` 는 동일한 Movie 정보를 이용하여, `MovieService`의 `createMovie`를 두번 호출한다. 두번째 호출시에 이미 등록된 Movie 정보이므로 `MovieException`이 발생하게 된다. 따라서 catch 블록의 `TransactionStatus`를 이용하여 현재 Transaction에서 발생한 변경 사항이 rollback 처리 된다. 다음은 테스트 클래스 `ProgrammaticTransactionManagementTest`의 `testAddMovieUsingTransactionTemplate` 메소드의 로직이다.

```
public void testAddMovieUsingTransactionTemplate() throws Exception {
    @Inject
    private TransactionTemplate transactionTemplate;

    @Inject
    private PlatformTransactionManager transactionService;

    transactionTemplate.execute(new TransactionCallbackWithoutResult() {
        public void doInTransactionWithoutResult(TransactionStatus status) {

            try {
                //1. Set Movie
                Movie newMovie = getMovie();
                newMovieID = newMovie.getMovieId();

                //2. 무비 등록 요청
                movieService.create(newMovie);

                //3. 동일한 무비 등록 요청
                movieService.create(newMovie);
            } catch (Exception e) {
                // 4. 현재 Transaction에서 발생한 변경 사항 rollback 처리
                status.setRollbackOnly();
            }
        }
    });

    try {
        // 5. 무비 등록 처리 rollback 여부 확인
        movieService.get(newMovieID);

        // 6. rollback이 성공적으로 이루어진 경우 해당 무비는 미등록 상태임
        fail("fail to transaction management.");
    } catch (Exception e) {
        assertTrue("fail to rollback.", e instanceof Exception);
    }
}
```

8.2.2.TransactionManager를 직접 이용한 Transaction 관리

8.2.2.1.Configuration

다음은 Transaction 서비스의 속성을 정의한 XML(context-transaction.xml) 파일로, Transaction을 관리하는 실질적 역할을 수행하는 TransactionManager가 정의되어 있다.

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource">
    <ref bean="dataSource"/></property>
</bean>
```

Transaction 서비스의 속성 정의시 본 매뉴얼 >> Spring >> Transaction Management 을 참고하도록 한다.

8.2.2.2.Transaction 관리

Transaction 서비스를 직접 얻어온 후에 다음과 같이 try~catch 구문 내에서 Transaction 서비스를 이용하여, 적절히 begin, commit, rollback을 수행한다. 이 때, TransactionDefinition와 TransactionStatus 객체를 적절히 이용하면 된다.

```
...
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);
TransactionStatus status = transactionService.getTransaction(def);
try {
  // ... biz logic ...
  transactionService.commit(status);
}
catch (Exception ex) {
  transactionService.rollback(status);
  throw ex;
}
...
```

8.2.2.3.테스트 클래스 실행

테스트 클래스 ProgrammaticTransactionManagementTest 는 동일한 Movie 정보를 이용하여, MovieService의 create를 두번 호출한다. 두번째 호출시에 이미 등록된 Movie 정보이므로 MovieException이 발생하게 된다. 따라서 catch 블록의 TransactionStatus를 이용하여 현재 Transaction에서 발생한 변경 사항이 rollback 처리 된다. 다음은 테스트 클래스 ProgrammaticTransactionManagementTest의 testAddMovieUsingTransactionManager 메소드의 로직이다.

```
public void testAddMovieUsingTransactionManager() throws Exception {
  DefaultTransactionDefinition txDefinition = new DefaultTransactionDefinition();
  // 해당 Transaction을 위한 Propagation Behavior, Isolation Level 등 정의
  txDefinition
    .setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);
  TransactionStatus status = transactionService
    .getTransaction(txDefinition);
  String newMovieId = "";
```

```

try {
    //1. Set Movie
    Movie newMovie = getMovie();
    newMovieId = newMovie.getMovieId();

    //2. 무비 등록 요청
    movieService.create(newMovie);

    //3. 동일한 무비 등록 요청
    movieService.create(newMovie);

    //4. 정상적으로 처리된 경우 현재 Transaction에서 발생한 변경 사항 commit 처리
    transactionService.commit(status);
} catch (Exception e) {
    transactionService.rollback(status);
}

try {
    // 6. 사용자 등록 처리 rollback 여부 확인
    movieService.get(newMovieId);

    // 7. rollback이 성공적으로 이루어진 경우 해당 사용자는 미등록 상태임
    fail("fail to transaction management.");
} catch (Exception e) {
    assertTrue("fail to rollback.", e instanceof Exception);
}
}

```

8.3.Resources

- 다운로드

다음에서 테스트 DB를 포함하고 있는 hsqldb.zip과 sample 코드를 포함하고 있는 anyframe-sample-transaction.zip 파일을 다운받은 후, 압축을 해제한다. 그리고 hsqldb 폴더 내의 start.cmd (or start.sh) 파일을 실행시켜 테스트 DB를 시작시켜 놓는다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, mvn compile exec:java -Dexec.mainClass=...이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, src/main/java 폴더의 anyframe/sample/transaction 하위의 Main.java를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 Run As > Java Application을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
hsqldb.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/hsqldb.zip]
anyframe-sample-transaction.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-transaction.zip]

- 참고자료

- Transaction management manual [<http://static.springsource.org/spring/docs/4.0.x/spring-framework-reference/html/transaction.html>]
- Bringing Advanced Transaction Management Capabilities to Spring Applications [http://www.oracle.com/technology/tech/java/spring/jta_spring_article.pdf]

III.Spring MVC

Spring MVC는 MVC(Model, View, Controller) 패턴 기반의 웹 프레임워크이다. 또한 Spring MVC는 Controller, Handler Mappings, ModelAndView, View Resolver, View 등의 구성 요소를 가지며, 모든 요청을 받아 요청을 처리할 Controller에게 전달해 주는 Front Controller로 DispatcherServlet을 사용한다. 실제로 요청을 처리하는 Controller(Request Handler)는 특정 인터페이스를 구현하거나 특정 클래스를 상속받지 않고도, @Controller와 @RequestMapping만을 사용하여 구현이 가능하기 때문에 이전 보다 다양한 형식으로 개발할 수 있다. 또한 Spring 3.x 부터 소개된 @PathVariable, @RequestBody, @ResponseBody 등의 Annotation 및 여러가지 새로운 특징들을 기반으로 RESTful 웹 어플리케이션을 구축할 수 있다.

Spring MVC에서는 어떤 Framework API에도 종속되지 않은 POJO를 입력 폼 데이터를 전달하기 위한 객체로 사용할 수 있다. 또한 Spring MVC는 서버에서 처리된 정보를 보여주기 위한 View를 결정하는 방식이 유연하기 때문에, 일반 JSP 뿐만 아니라 Tiles, Velocity 등 다양한 View 기술과 쉽게 연계가 가능하다.

Spring MVC 웹 프레임워크는 다음과 같은 특징을 가진다.

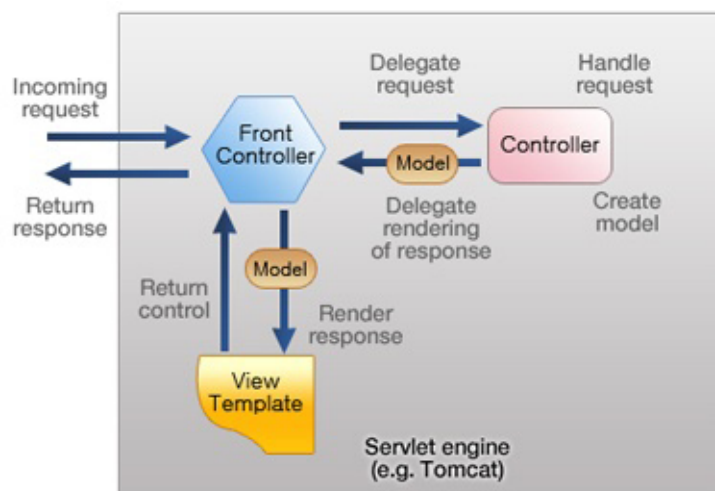
- 역할 분리가 명확하다. controller, validator, command 객체, 폼 객체, model 객체, DispatcherServlet, handler mapping, view resolver 등의 각각의 역할은 해당 역할만을 전문으로 수행하는 객체들이 담당한다.
 - 어플리케이션 내의 JavaBean들과 프레임워크에 관련된 설정이 쉽고 간단하다.
 - Business 객체를 Framework에 종속된 API를 사용하여 확장하지 않고도 command 또는 폼 객체로 재사용할 수 있다.
 - Application 레벨에서 데이터를 바인딩 하고 validation 에러를 체크할 수 있도록 데이터 바인딩 및 검증을 customizing 할 수 있다.
 - 간단한 URL 기반 설정으로 다양한 handler mapping과 view resolution을 customizing 할 수 있다.
 - 모델이 맵으로 구성되기 때문에 여러 view 기술과의 연계가 쉽다.
 - 데이터 바인딩이나 테마 사용을 위한 spring 태그를 제공한다.
 - JSP의 입력 폼을 보다 쉽게 만들 수 있는 form 태그를 제공한다.
-

9. Architecture

Spring MVC는 MVC 패턴 기반의 Model2아키텍처를 사용하며 Model, View, Controller 컴포넌트로 구성된다.

- **Model** : Spring MVC에서는 Model 컴포넌트를 만드는 방법을 직접 제공하지 않는다. 대신 EJB (Enterprise Java Beans), JDO (Java Data Objects), JavaBeans, ORM (Object to Relational Mapping framework) 등 여러 기술들을 이용해 구현된 어떤 Model 컴포넌트든 접근 가능하다. 또한 폼 입력 필드 값을 별도로 폼 객체를 개발할 필요 없이 일반 모델 객체로 바인딩 할 수 있는데, 이 때 모델 객체 attribute로 자동으로 매핑되어 정의된 타입에 맞게 타입 변환이 이루어진다. 단, attribute명과 입력 필드 명이 일치해야한다.
- **View** : 표준 JSP 나 Spring MVC에서 제공하는 tag library를 사용하여 View 컴포넌트를 제작한다. Spring MVC에서는 별도의 bean, html, logic 태그는 제공하지 않으며 표준JSP 태그인 JSTL을 사용할 것을 권장한다. 컴포넌트의 재사용, 관리 노력의 절감, 에러 최소화를 위해 Application-Specific Custom tag, Image Rendering Component 등 다른 기술의 채택을 고려할 수 있다.
- **Controller** : Spring MVC에서는 서버로 입력된 요청을 실제로 처리하는 Controller(Spring MVC에서는 Handler라고 일컫는다.)를 쉽게 개발할 수 있도록 다양한 Annotation들을 제공하고 있다. Spring MVC의 DispatcherServlet은 모든 요청을 받아서 해당 요청을 처리할 Controller로 전달해 주는 "Front Controller" 역할을 담당하고 있다.

FrontController 역할을 하는 DispatcherServlet의 요청 처리 workflow는 아래의 그림과 같다.



위의 그림에서 볼 수 있듯이 모든 요청이 통과하는 곳은 Front Controller이며 Spring MVC에서는 DispatcherServlet이 이 Front Controller 역할을 한다. DispatcherServlet은 모든 요청을 받아서, Locale, Theme, Multipart 등과 관련된 작업을 처리한 후 HandlerMapping을 통해 각각의 요청을 처리할 Handler를 찾아내어 요청을 전달한다. Handler서 요청을 처리한 뒤 View정보와 응답으로 보여줄 정보를 포함한 ModelAndView 객체를 다시 DispatcherServlet에게 리턴하면 DispatcherServlet은 ModelAndView 객체와 ViewResolver를 통해 View를 찾고 해당 View를 통해 응답을 전달한다.

10.Configuration

먼저, Spring MVC 기반의 웹 어플리케이션을 개발하기 위해서 반드시 정의되어야 하는 설정들을 알아보자. 하나는 Front Controller 역할을 하는 DispatcherServlet 등록을 위한 web.xml 파일이고 다른 하나는 Spring MVC의 구성 요소를 정의하기 위한 [servlet-name]-servlet.xml(예. action-servlet.xml) 파일이다.

10.1.web.xml 작성

JavaEE 웹 어플리케이션은 반드시 WEB-INF 디렉토리 하위에, 배포 기술서인 web.xml 파일이 존재해야 한다. Spring MVC를 이용한 웹 어플리케이션을 개발하기 위해서는 이 web.xml 파일에 DispatcherServlet을 등록하고, Spring MVC 기반의 웹 어플리케이션 구성요소들이 정의되어 있는 XML 설정 파일의 위치를 지정해주어야 한다. 작성 방법은 아래와 같다.

10.1.1.DispatcherServlet 등록

다음은 web.xml 파일에 DispatcherServlet을 정의한 모습이다.

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

위와 같이 정의할 경우 서블릿의 이름은 'action'이 된다. DispatcherServlet은 디폴트로 '[servlet-name]-servlet.xml' 파일을 WebApplicationContext로 로드하므로, Web Contents Root 폴더 아래의 WEB-INF 폴더에서 'action-servlet.xml' 파일을 찾아 WebApplicationContext를 구성하게 될 것이다.

또한 위의 서블릿 매핑 설정으로 인해서 URL의 확장자가 ".do"인 모든 URL에 대한 요청은 DispatcherServlet이 처리하게 된다.

10.1.2.Spring MVC 설정 파일 위치 등록

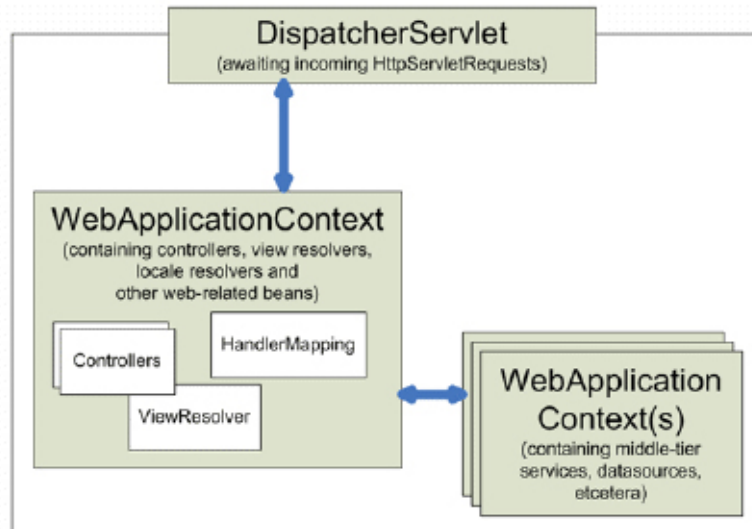
위에서 언급하였듯이 DispatcherServlet은 기본적으로 Web Contents Root 폴더 하위의 WEB-INF 폴더에 있는 [servlet-name]-servlet.xml 파일을 로드하여 WebApplicationContext를 구성하게 되는데, 이를 임의의 위치에 존재하는 임의의 이름을 가진 파일 또는 다중의 파일로 정의하기 위해서는 아래와 같이 <servlet> 하위에 <init-param>을 이용하여 contextConfigLocations 라는 초기화 파라미터를 정의해 준다.

```
<init-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/config/springmvc/common-servlet.xml, /config/springmvc/user-servlet.xml</param-value>
</init-param>
```

10.2.action-servlet.xml 작성

Spring MVC 프레임워크에서 각각의 DispatcherServlet은 고유의 WebApplicationContext를 가지고 있고, 각각의 WebApplicationContext는 상위의 WebApplicationContext에 정의된 모든 Bean 정보를 상속받아

사용할 수 있다. 이와 같이 `WebApplicationContext`는 계층구조를 가질 수 있는데, 그 모습을 Spring의 reference 문서에서는 다음 그림과 같이 표현하고 있다.



`WebApplicationContext`는 웹 어플리케이션에 필요한 몇몇 기능들을 추가하여 `ApplicationContext`를 확장한 것으로 Spring의 IoC 컨테이너의 한 종류이다. `WebApplicationContext`에는 웹 어플리케이션을 구성하고 있는 여러가지 Bean들이 등록되어 관리된다. 필요한 경우 `RequestContextUtils` 클래스를 통해 `WebApplicationContext`를 직접 록업해서 사용할 수도 있다. Spring MVC의 `DispatcherServlet`은 Request를 처리하고 적절한 View를 전달해 주기 위해서 아래 표와 같은 특별한 Bean들을 사용하는데, 이러한 Bean들을 일반 Bean처럼 `WebApplicationContext`에 설정할 수 있다.

Bean type	설명
Controllers	요청을 처리하는 컨트롤러들
Handler mappings	요청된 URL과 해당 URL을 처리할 컨트롤러와의 매칭을 처리
View resolvers	View 이름을 이용해 View를 결정
Locale resolver	국제화(i18n) 지원을 위해 사용자의 locale 알아냄
Theme resolver	웹 어플리케이션이 사용하는 테마를 결정
Multipart file resolver	HTML 폼으로 부터 업로드된 파일을 처리하는 기능을 가짐
Exception resolver	특정 예외와 각각의 예외에 맞는 view를 매핑하는 기능을 가짐

위와 같은 Bean들을 `action-servlet.xml` 파일에 정의하여 사용하게 된다.

`web.xml` 설정이 끝나면 위에서 설명한 특별한 Bean들을 `action-servlet.xml` 파일에 정의해주어야 한다. 이 장에서는 위 표의 요소들 중 Handler Mapping과 View Resolver를 정의하는 방법에 대해만 알아보고 다른 요소 (Controller, Locale Resolver, Multipart File Resolver, Exception Resolver)들에 대해서는 각각의 상세 페이지에서 설명하도록 한다.

10.2.1.Handler Mapping

Front Controller인 `DispatcherServlet`으로 요청이 들어왔을 때, 그 요청을 실제로 어떤 Controller가 처리할 것인지는 Handler Mapping을 통해서 알아낼 수 있다. Spring MVC에서는 여러가지 Handler Mapping을 제공하는데, 그 중 대표적인 것들만 알아보도록 하겠다.

- `RequestMappingHandlerMapping`

RequestMappingHandlerMapping은 Spring 3.1 버전부터 DefaultAnnotationHandlerMapping을 대체하여 제공되는 Class로, @RequestMapping annotation을 기반으로 구현된 Controller와 요청 URL을 매핑시켜주는 HandlerMapping 구현클래스이다. Java 5 이상인 경우 DispatcherServlet이 디폴트로 등록해준다. 따라서 사용자가 명시적으로 정의할 필요가 없지만, RequestMappingHandlerMapping 클래스가 가진 디폴트 속성들을 변경하고자 할때는 action-servlet.xml에 명시적으로 정의할 수 있다. 다음은 사용자가 변경할 수 있는 RequestMappingHandlerMapping의 속성들이다.

Property	설명
interceptors	사용할 interceptor들의 목록
defaultHandler	요청을 처리할 Controller를 찾지 못했을 때 디폴트로 사용할 Controller
order	여러가지 Handler mapping을 사용할 경우 order 속성에 정의된 값을 기반으로 순서대로 동작
alwaysUseFullPath	이 속성의 값이 true인 경우, servlet context 하위의 전체 URL path를 가지고 요청을 처리할 Controller를 찾고, 디폴트 값인 false인 경우 DispatcherServlet과 mapping한 URL path 하위의 path로 요청을 처리할 Controller를 찾는다. 예를 들어, 현재 DispatcherServlet이 '/rest/*'와 매핑되어있고, 이 속성값이 true로 셋팅되어 있다면, '/rest/welcome.jsp' 전체가 사용되고, false인 경우 'welcome.jsp'만 사용될 것이다.

다음은 interceptors 속성을 오버라이드하여 RequestMappingHandlerMapping을 정의한 예이다.

```
<beans>
  <bean
    class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">
    <property name="interceptors">
      <bean class="example.MyInterceptor"/>
    </property>
  </bean>
</beans>
```

- DefaultAnnotationHandlerMapping

DefaultAnnotationHandlerMapping은 Spring 3.0 이하 버전에서 제공되던 HandlerMapping 구현클래스로, Spring 3.1 버전부터는 위에서 소개한 RequestMappingHandlerMapping 클래스를 기본으로 등록하여 사용 하고 있다. 기본적인 사용법은 RequestMappingHandlerMapping 클래스와 동일하다.

- BeanNameUrlHandlerMapping

BeanNameUrlHandlerMapping은 요청 URL과 정의된 Controller Bean의 이름을 비교하여 해당 요청을 처리할 Controller를 매핑해준다. 다음은 BeanNameUrlHandlerMapping을 정의한 action-servlet.xml 파일의 일부이다.

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
```

```
<bean name="/login.do"
  class="org.anyframe.sample.springmvc.web.controller.basic.LoginController"></bean>
```

action-servlet.xml에 위와 같이 정의되어 있는 경우, "/login.do"라는 요청이 들어왔을 때, org.anyframe.sample.springmvc.web.controller.basic.LoginController가 처리하게 된다.

- SimpleUrlHandlerMapping

SimpleUrlHandleMapping은 요청 URL과 요청을 처리할 Controller간의 매핑 정보를 하나의 저장소에서 관리할 수 있도록 해준다. 사용자는 Controller를 일반 Bean으로 정의해 주고, SimpleUrlHandleMapping의 mappings 속성에 요청 URL과 요청을 처리할 Controller Bean의 ID를 정의한다. 다음은 위의 BeanNameUrlHandlerMapping 예시를 SimpleUrlHandlerMapping을 이용해 나타낸 것이다.

```
<bean id="simpleUrlHandlerMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>/login.do = loginController</value>
    </property>
</bean>
<bean id="loginController"
    class="org.anyframe.sample.springmvc.web.controller.basic.LoginController"></bean>
```

또한 SimpleUrlHandlerMapping을 사용할 경우 매핑 정보를 빈 설정 파일이 아닌 별도의 파일에서 관리하는 것이 가능하다. 예는 다음과 같다.

```
<bean id="simpleUrlHandlerMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <bean class="org.springframework.beans.factory.config.PropertiesFactoryBean">
            <property name="location">
                <value>/mapping.properties</value>
            </property>
        </bean>
    </property></bean>
```

다음은 위에서 정의된 mapping.properties파일의 내용이다.

```
/login.do = loginController
```

- Intercepting requests

handler mapping에는 interceptor를 정의할 수 있으며 해당 handler mapping에 의해 처리되는 요청은 정의한 interceptor가 적용되게 된다. 이러한 interceptor는 요청을 가로채서 요청이 들어오기 전, 들어온 후, 완료된 후에 특정 작업을 추가할 수 있다. interceptor 클래스는 org.springframework.web.servlet.HandlerInterceptorAdapter 클래스를 상속받아 생성하고 preHandle(), postHandle(), afterCompletion() 메소드를 구현하여 각 시점에 따라 처리 로직을 추가할 수 있다.

다음은 LoginInterceptor.java 파일의 일부이다.

```
public class LoginInterceptor extends HandlerInterceptorAdapter {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
        Object handler) throws Exception {
        if(request.getSession().getAttribute("userId") != null)
            return true;
        else {
            response.sendRedirect("login.jsp");
            return false;
        }
    }
}
```

위의 예에서는 preHandle() 메소드를 오버라이딩 하여 요청이 들어오기 전에 해당 로직을 수행하게 된다. session에 userId 값이 존재할 경우 true를 리턴하고 이어서 Controller가 요청을 처리하게 될 것이고, userId 값이 존재 하지 않는다면 login.jsp 페이지를 출력하게 될것이다.

다음은 빈 설정파일에 interceptor를 설정한 user-servlet.xml 파일의 일부이다.

```
<bean id="simpleUrlHandlerMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>/userForm.do = userController</value>
    </property>
    <property name="interceptors" ref="loginInterceptor"/>
    <!-- 여러 개의 handler mapping이 정의되어 있을 시에 order를 정의하여 우선순위를 부여할 수 있다.
        숫자가 작을수록 높은 우선순위를 갖는다.-->
    <property name="order">
        <value>1</value>
    </property>
</bean>

<bean id="loginInterceptor"
    class="org.anyframe.sample.springmvc.web.interceptor.LoginInterceptor" />
```

위와 같이 interceptor 클래스를 빈으로 설정하고 handler mapping에서 해당 빈을 참조하여 interceptor를 적용시킬 수 있다.

10.2.2.View Resolver

모든 MVC Framework에서는 요청을 처리한 후 돌아갈 View를 지정하기 위한 방법을 제공한다. Spring MVC에서는 특정 View 기술에 종속되지 않고 Model 데이터들을 보여줄 수 있도록 View Resolver를 제공한다. 앞에서 설명했듯이 핸들러(Controller)는 요청을 처리한 뒤 다시 DispatcherServlet에게 ModelAndView 객체를 넘겨준다. 이 때 ModelAndView는 View의 이름을 포함하고 있는데, 이 이름으로 실제 View를 찾아주는 역할을 하는 것이 View Resolver이다.

ViewResolver [<http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/web/servlet/ViewResolver.html>]와 View [<http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/web/servlet/View.html>]는 Spring MVC에서 View 처리와 관련된 가장 중요한 인터페이스이다. ViewResolver는 View 이름과 실제 View를 매핑해준다. Spring MVC에서 제공하는 View Resolver에는 다음과 같은 것들이 있다.

ViewResolver	설명
AbstractCachingViewResolver	View들을 caching하는 기능 제공
XmlViewResolver	View를 결정할 때 XML 파일의 설정 내용을 기반으로 판단 (/WEB-INF/view.xml을 기본 설정 파일로 사용)
ResourceBundleViewResolver	View를 결정할 때 리소스 파일의 설정 내용을 기반으로 판단 (views.properties를 기본 리소스 파일로 사용)
UrlBasedViewResolver	View를 결정할 때 특정 �핑 정보를 사용하지 않고, View 이름으로 URL을 사용(View 이름과 실제 View 자원을 동일하게 사용하고자 할 때 사용)
InternalResourceViewResolver	UrlBasedViewResolver를 상속 받았으며 InternalResourceView(Servlet, JSP)를 사용
VelocityViewResolver/FreeMarkerViewResolver	UrlBasedViewResolver를 상속 받았으며 각각 VelocityView와 FreeMarkerView를 사용

ViewResolver	설명
ContentNegotiatingViewResolver	요청 URL의 파일명이나 HTTP Request의 Accepter 헤더값을 기반으로 View를 판단. 자세한 사용법은 본 매뉴얼의 SpringREST Plugin >> Content Negotiation [http://dev.anyframejava.org/docs/anyframe/plugin/optional/springrest/1.1.0/reference/htmlsingle/springrest.html#springrest_restsupport_contentnegotiation]을 참조

사용하려는 View 기술에 따라 위와 같은 View Resolver를 적절히 선택해야 한다.

- JSP를 View 기술로 사용할 경우 ViewResolver 설정 예

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
```

UrlBasedViewResolver에는 prefix와 suffix 속성을 지정해 줄 수 있다. 만약 Controller에서 넘겨준 View 이름이 'index'이고 prefix를 "/jsp/", suffix를 ".jsp"라고 정의했다면 이 ViewResolver는 "/jsp/index.jsp"라는 이름의 View를 찾아준다.

- JSTL 사용 시의 ViewResolver 설정 예

단순 JSP인 경우 UrlBasedViewResolver는 InternalResourceView를 사용하지만, JSTL을 사용할 경우에는 다음과 같이 viewClass 속성을 통해 JstlView를 사용하도록 명시적으로 정의해주어야 한다.

```
<bean id="jstlViewResolver"
    class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <!-- view class for jstl -->
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
    <property name="order" value="1" />
</bean>
```

10.2.3.Configuration Simplification

Spring 3에서는 Annotation 기반의 Controller 처리를 위해 반드시 필요한 RequestMappingHandlerAdapter 등록 등의 Spring MVC 관련 설정을 간편하게 할 수 있도록 mvc [<http://static.springsource.org/schema/mvc/spring-mvc.xsd>] 네임스페이스를 제공하기 시작했다. mvc 네임스페이스에서 제공하는 태그는 다음과 같이 3가지가 있다.

- **mvc:annotation-driven**

Annotation 기반으로 구현된 Controller로 HTTP 요청을 처리하기 위해서 필요한 클래스들을 쉽게 등록해주는 태그이다. 이 태그는 HTTP 요청을 읽어 실행시킬 Controller - Method를 선택(select)하는 RequestMappingHandlerMapping, 선택된 Method를 실행(involve)시키는 RequestMappingHandlerAdapter, Controller 내에서 발생한 예외를 처리하기 위한 ExceptionHandlerExceptionResolver를 bean으로 등록한다. 기본으로 등록되는 3개의 클래스 외에 다음의 몇 가지 설정들이 기본으로 등록된다.

1. Spring 3의 Type ConversionService를 사용할 수 있도록 org.springframework.format.support.FormattingConversionServiceFactoryBean에 의해 생성된 ConversionService 인스턴스를 등록해준다. ConversionService를 변경하고자 하는 경우 **conversion-service** 속성을 사용하여 설정해준다.
2. @NumberFormat을 사용한 Number 타입의 formatting 지원

3. @DateTimeForm을 사용한 Date, Calendar Joda Time 타입의 필드의 formatting 지원 (단, 현재 classpath에 Joda Time 라이브러리가 존재할 경우)
4. Annotation 기반으로 구현된 Controller에서 @Valid를 사용한 선언적인 Validation 기능 지원 (단, 현재 classpath에 Hibernate Validator와 같은 JSR-303을 구현체가 존재하는 경우)
5. XML의 Read/Write 지원 (단, 현재 classpath에 JAXB 라이브러리가 존재하는 경우)
6. JSON 객체의 Read/Write 지원 (단, 현재 classpath에 Jackson [<http://jackson.codehaus.org/>] 라이브러리가 존재하는 경우)

```
<mvc:annotation-driven conversion-service="conversionService" />

<bean id="conversionService"
      class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="org.anyframe.sample.moviefinder.StringToFilmRatingConverter" />
      <bean class="org.anyframe.sample.moviefinder.FilmRatingToStringConverter" />
    </list>
  </property>
</bean>
```



Spring @MVC 3.1 new infrastructure

Spring 3.1에서는 보다 쉽고 다양한 설정을 사용할 수 있도록 다음 세 개의 새로운 Class를 제공한다.

- RequestMappingHandlerMapping
- RequestMappingHandlerAdapter
- ExceptionHandlerExceptionHandlerResolver

위의 Class들은 기존에 존재하던 아래의 Class를 대체한다.

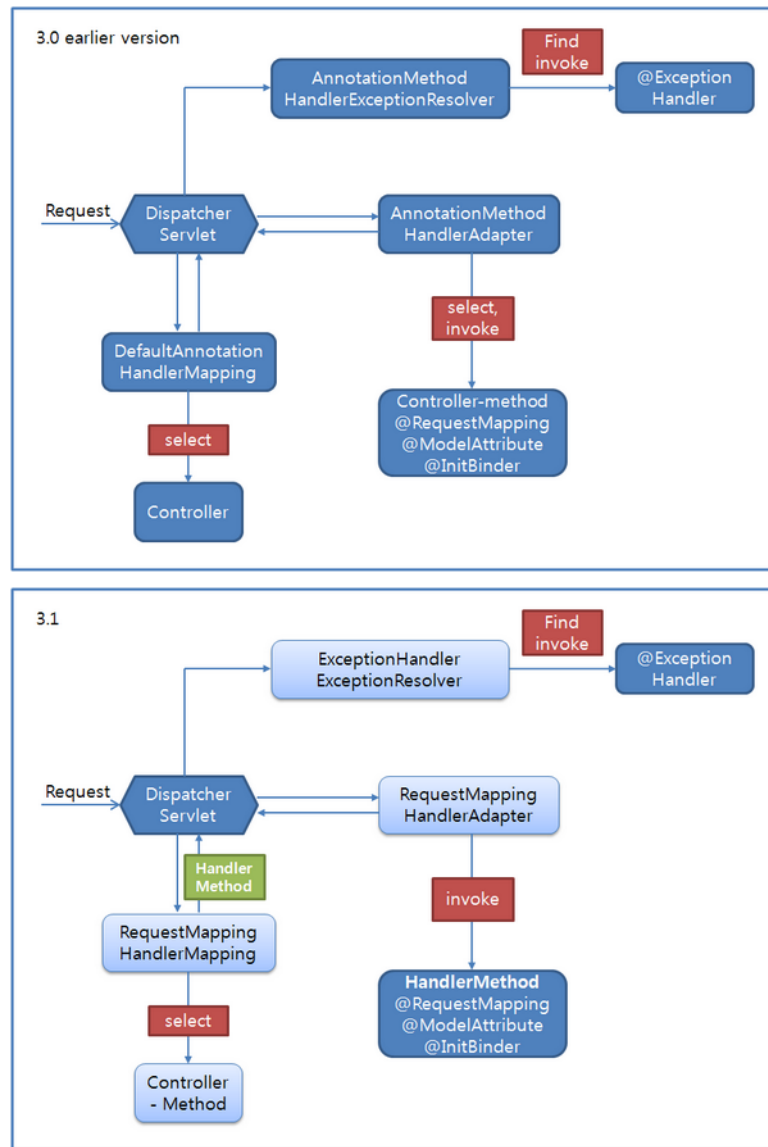
- DefaultAnnotationHandlerMapping
- AnnotationMethodHandlerAdapter
- AnnotationMethodHandlerExceptionHandlerResolver

Spring 3.1의 mvc:annotation-driven 설정을 등록하게 되면 기존의 DefaultAnnotationHandlerMapping, AnnotationMethodHandlerAdapter를 대신해 RequestMappingHandlerMapping, RequestMappingHandlerAdapter, ExceptionHandlerExceptionHandlerResolver가 기본으로 등록된다.

Spring 3.0 이하의 버전에서는 실제 수행 될 Controller의 Method를 찾아서 실행 시키기 위해 2번의 select, 1번의 invoke 동작을 내부적으로 수행 하고 있었다.

Spring 3.1에서는 RequestMappingHandlerMapping class가 Controller의 Method를 select하고, RequestMappingHandlerAdapter가 선택 된 Method를 invoke하도록 구조가 변경 되었다.

다음은 기존의 class와 새로 제공되는 class간의 차이점을 그림으로 나타낸 것이다.



mvc:annotation-driven 설정을 사용하거나, @EnableWebMvc Java-based configuration 설정을 사용한 경우 앞서 살펴본 새로운 class들이 기본값으로 설정되며, 기존의 class도 여전히 사용 가능하지만 새로 제공되는 class를 사용할 것을 권장한다.

- **mvc:interceptors**

Spring 2.x까지는 요청 처리 전에 특정 기능 수행을 목적으로 Interceptor를 적용하기 위해서는 각각의 HandlerMapping Bean 정의 시에 interceptors 속성을 이용하여 적용할 Interceptor를 추가해 주어야만 했다. 그러나 Spring 3 부터는 <mvc:interceptors>를 사용하면 모든 HandlerMapping에 Interceptor를 간편하게 적용할 수 있다.

다음은 모든 URL에 LocaleChangeInterceptor를 적용한 예이다.

```
<mvc:interceptors>
  <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />
</mvc:interceptors>
```

특정 URL에만 Interceptor를 적용하려면 아래와 같이 설정해주면 된다.

```
<mvc:interceptors>
  <mvc:interceptor>
    <mapping path="/admin/*"/>
```

```
<bean class="sample.LoginInterceptor" />
</mvc:interceptor>
</mvc:interceptors>
```

- **mvc:view-controller**

이 태그는 Spring 3 이전에 Controller 없이 바로 View로 포워딩하는 URL에 대해서 ParameterizableViewController를 사용하여 Bean으로 정의했던 것을 아래와 같이 간편하게 설정할 수 있도록 해준다.

```
<mvc:view-controller path="/" view-name="welcome"/>
```

11.Controller

MVC에서 C에 해당하는 컨트롤러는 사용자의 요청을 받아서 어플리케이션에 정의된 적절한 Service를 수행한 후, 그 결과를 다시 View를 통해 사용자에게 보여줄 수 있는 Model 데이터로 변환하는 역할을 담당한다. Spring에서는 이러한 컨트롤러를 특정 API에 종속되지 않고 사용자가 자유롭게 작성할 수 있는 추상적인 구현 방법을 제공하고 있다.

Spring 2.5에서부터 @RequestMapping, @RequestParam, @ModelAttribute 등을 이용한 Annotation 기반의 컨트롤러 개발 방식을 소개했다. Annotation을 사용하여 SpringMVC기반의 컨트롤러를 작성하면, 특정 인터페이스를 상속받거나 특정 클래스를 상속받지 않아도 된다. 또한 Servlet API와도 독립적으로 작성할 수 있다는 장점이 있다. (단, annotation은 JAVA 5 이상에서만 사용가능함에 유의하도록 한다.)



Spring MVC controller hierarchy deprecated

기존에 Spring에서 제공하던 AbstractFormController등의 Form 관련 Class 계층은 Spring 3 부터는 더이상 지원하지 않는다. Spring MVC에서는 @Controller, @RequestMapping 등의 Annotation을 기반으로 컨트롤러를 개발하도록 가이드하고 있다.

11.1.Configuration

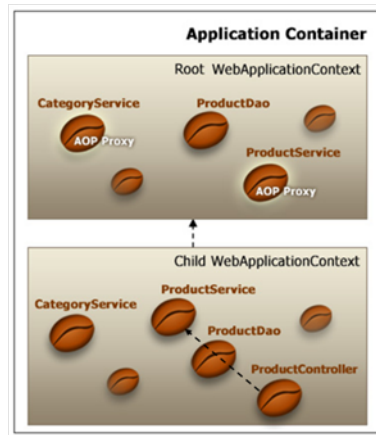
컨트롤러 역할을 수행하는 클래스를 정의하기 위해서는 Spring에서 제공하는 Stereotype Annotation 중 @Controller를 사용한다. 이렇게 정의된 컨트롤러 클래스는 XML 설정 파일에 명시적으로 Bean으로 정의하여 Spring Container에 등록할 수도 있지만, 자동으로 검색 및 등록이 가능하게 할 수도 있다. Spring에서는 이를 Autodetection이라고 한다. **Stereotype Annotation이 적용된 클래스들에 대한 Autodetection이 이루어 지도록 하기 위해서는 <context:component-scan/> 을 속성 정의 XML에 추가해 주어야 한다.** <context:component-scan/>에 대한 자세한 내용은 본 매뉴얼 >> Spring >> Annotation을 참고하기 바란다.

11.1.1.Using Filters to customize scanning

<context:component-scan/>은 해당 클래스스페이스 내에 @Component, @Service, @Repository, @Controller가 적용된 클래스를 모두 찾아서 Spring 컨테이너가 관리하는 컴포넌트로 등록하도록 하는 설정이다. 이와 같은 디폴트 동작 방식으로 Autodetection 기능 이용 시, 비즈니스와 프레젠테이션 레이어 간 Bean 정의 XML을 분리하여 관리하면서 <context:component-scan/>을 중복으로 설정하는 경우 다음과 같은 문제가 발생할 수 있다.

- Autodetection 중복 설정으로 인해 야기되는 문제점
 - Stereotype Annotation이 적용된 클래스가 비즈니스 레이어의 Root WebApplicationContext와 프레젠테이션 레이어의 WebApplicationContext에 중복하여 등록된다.
 - 비즈니스 레이어의 Root WebApplicationContext와 프레젠테이션 레이어의 WebApplicationContext는 Parent Container - Child Container 관계로 구성된다. Container가 계층 구조를 가질 때, 사용하고자 하는 Bean 검색 순서는 현재 자기 Container가 먼저이고, Bean이 없을 경우 Parent Container가 그 다음이다.
 - 일반적으로 AOP 설정은 비즈니스 레이어에서 관리한다. 따라서 Proxy 기반의 Spring AOP는 비즈니스 레이어의 Root WebApplicationContext에 등록된 Bean에만 적용되고, 프레젠테이션 레이어의 WebApplicationContext에 중복으로 등록된 Bean에는 적용되지 않는다.
 - 결과적으로 프레젠테이션 레이어의 WebApplicationContext에서는 Proxy 기반의 Spring AOP가 적용되지 않은 Bean을 먼저 참조하게 되어 Spring AOP를 사용하여 설정한 기능들이 동작하지 않는 문제점이 발생한다.

다음은 위의 내용을 그림으로 나타낸 것이다.



이와 같은 문제를 방지하기 위해서 비즈니스 레이어(Root WebApplicationContext)에서 관리되어야 하는 Bean과 프레젠테이션 레이어(Child WebApplicationContext)에서 관리되어야 하는 Bean을 구분할 필요가 있다.

다음은 프레젠테이션 레이어에서 **@Controller** annotation이 적용된 클래스만 **WebApplication Context**에 등록하는 common-servlet.xml 파일의 설정 예이다.

```
<!-- use-default-filters="false"로 설정하고
include-filter를 사용했기 때문에 이 WebApplicationContext에는 @Controller가 적용된 클래스만
등록된다. -->
<context:component-scan base-package="org.anyframe.sample.springmvc" use-default-
filters="false">
    <context:include-filter type="annotation"
        expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

위의 예와 같이 <context:component-scan>하위에 <context:include-filter> 나 <context:exclude-filter>를 추가하면 컨테이너에 의해 검색될 대상의 범위를 조정할 수 있다. filter에 대한 자세한 내용은 본 매뉴얼 >> Spring >> Annotation 을 참고 바란다.

11.2.컨트롤러 구현

앞에서 설명했듯이, Spring MVC에서는 요청을 처리하는 컨트롤러를 특정 인터페이스 구현하거나 특정 클래스 상속받아서 구현하지 않아도 된다. @Controller, @RequestMapping 등의 Annotation만을 이용하여 다양한 형태의 컨트롤러를 만들 수 있다. 본 문서에서는 Spring MVC에서 제공하는 Annotation을 사용하여 컨트롤러를 작성하는 방법에 대해서 알아본다.

- @Controller : 컨트롤러 클래스 정의
- @RequestMapping : 처리할 HTTP Request URL과 컨트롤러 클래스 또는 메소드 매핑
- @RequestParam : HTTP Request에 포함된 파라미터 참조 시 사용
- @RequestHeader : HTTP Request의 Header 값 참조 시 사용
- @CookieValue : HTTP Cookie 값 참조 시 사용
- @ModelAttribute : HTTP Request에 포함된 파라미터를 Model 객체로 바인딩함, @ModelAttribute의 'name'으로 정의한 Model객체를 다음 View에서 사용 가능
- @SessionAttributes : Session에 저장할 Model attribute를 정의
- @RequestBody/@ResponseBody : 핸들러 메소드가 HTTP Request와 Response의 Body 메시지 전체를 직접 접근할 경우에 사용 가능. (HttpEntity 객체를 이용하여 HTTP Request나 Response의 Body 메시지나 Header 값을 처리할 수도 있다.)

- @RestController : @Controller 와 @ResponseBody 를 합친 동작을 하는 Annotation. Stereotype Annotation으로, componenet scan을 통해 bean으로 등록해서 사용 할 수 있다.

11.2.1.@Controller

특정 클래스에 @Controller annotation을 적용하면 다른 클래스를 상속받거나 Servlet API를 사용하지 않아도 해당 클래스가 컨트롤러 역할을 수행하도록 정의할 수 있다.

다음은 @Controller를 사용하여 작성한 MovieController 클래스 파일의 일부이다.

```
@Controller
public class MovieController {
    // 중략
}
```

11.2.2.@RequestMapping

@RequestMapping annotation은 컨트롤러 클래스나 메소드가 특정 HTTP Request URL을 처리하도록 매핑하기 위해서 사용한다. 그래서 클래스 선언부에 @RequestMapping을 적용할 수도 있고(이하 Type-Level), 클래스의 메소드에 @RequestMapping을 적용할 수도 있다(이하 Method-Level). 예를 들어, Type-Level에 @RequestMapping("/movies")라고 정의하고, Method-Level에 @RequestMapping("/new") 라고 정의하면 @RequestMapping("/new")라고 정의한 메소드가 처리하는 URL 경로는 "/movies/new" 가 된다. @RequestMapping은 "/movies/*.do"와 같은 Ant 스타일 경로 패턴도 지원한다. @RequestMapping에는 URL 경로 외에도 HTTP method나 Request 파라미터 등을 추가하여 처리할 URL의 범위를 줄일 수 있다.

또한, Spring 3 부터 REST 스타일의 Web Application 개발을 위해서 URI templates을 지원하기 시작했다. Spring 3에서 추가된 REST 관련 기능들과 REST Style 웹 어플리케이션 개발에 대한 자세한 내용은 본 매뉴얼 Spring REST Plugin [http://dev.anyframejava.org/docs/anyframe/plugin/optional/springrest/1.1.0/reference/htmlsingle/springrest.html#springrest_restsupport_part]을 참고하기 바란다.

다음은 @RequestMapping을 사용하여 처리할 URL을 매핑한 코드예이다.

```
@Controller
@RequestMapping("/coreMovie.do")
public class MovieController {
    @RequestMapping(params="method=get")
    public String get(@RequestParam("movieId") String movieId, Model model) throws Exception
    {
        Movie movie = this.movieService.get(movieId);
        //...
        model.addAttribute(movie);
        return "coreViewMovie";
    }
}
```

@RequestMapping annotation에는 다음과 같은 상세 속성 정보를 부여하여 처리할 URL의 범위를 한정지을 수 있다.

name	Description
value	"value='/getMovie.do'"와 같은 형식의 매핑 URL 값이다. 디폴트 속성이기 때문에 value만 정의하는 경우에는 'value='은 생략할 수 있다. URL Pattern의 배열 형태로 입력 받으며, OR 조건으로 처리한다.

name	Description
	예 : @RequestMapping(value={"/addMovie.do", "/updateMovie.do" }) 이와 같은 경우 "/addMovie.do", "/updateMovie.do" 두 URL 모두 처리한다.
method	<p>GET, POST, HEAD 등으로 표현되는 HTTP Request method에 따라 requestMapping을 할 수 있다. 'method=RequestMethod.GET' 형식으로 사용한다. method 값을 정의하지 않는 경우 모든 HTTP Request method에 대해서 처리한다.</p> <p>HTTP Request method의 배열 형태로 입력 받으며, OR 조건으로 처리한다.</p> <p>예 : @RequestMapping(method = {RequestMethod.POST, RequestMethod.GET}) 이 경우, POST, GET 두 메소드 타입을 모두 처리한다. 또, value 값은 클래스 선언에 정의한 @RequestMapping의 value 값을 상속받는다.</p>
params	<p>HTTP Request로 들어오는 파라미터 표현이다. 'params={"param1=a", "param2", "!myParam"}' 로 다양하게 표현 가능하다.</p> <p>HTTP Request에 포함된 parameter의 배열 형태로 입력 받으며, AND 조건으로 처리한다.</p> <p>예 : @RequestMapping(params = {"param1=a", "param2", "!myParam"}) 위의 경우 HTTP Request에 param1과 param2 파라미터가 존재해야하고 param1의 값은 'a'이어야하며, myParam이라는 파라미터는 존재하지 않아야한다. 또한, value 값은 클래스 선언에 정의한 @RequestMapping의 value 값을 상속받는다.</p>
headers	<p>HTTP Request의 헤더 값이다. 'headers="someHeader=someValue"', 'headers="someHeader"', 'headers="!someHeader"' 로 다양하게 표현 가능하다. Accept나 Content-Type 같은 헤더에 대해서 media type 표현 시 '*' 도 지원한다. HTTP Request의 헤더값을 배열로 입력 받으며, AND 조건으로 처리된다. 단, 이때 Accept와 Content-type 조건은 제외된다. 자세한 내용은 아래의 consumes, produces 부분을 참고한다.</p> <p>예 : @RequestMapping(value="/movie.do", headers="Content-type=text/*") 의 경우 HTTP Request에 Content-Type 헤더 값이 "text/html", "text/plain" 모두 매칭이 된다. 또한, Type-Level, Method-Level에서 모두 사용할 수 있는데, Type-Level에 정의된 경우, 하위의 모든 핸들러 메소드에서도 Type-Level에서 정의한 헤더값 제한이 적용된다.</p>
consumes	<p>Spring 3.1에서부터 제공하는 기능으로, HTTP Request의 헤더 값 중에서 'Content-Type'으로 정의된 media type 표현식의 배열을 입력값으로 받으며, OR 조건으로 처리된다.</p> <p>예 : @RequestMapping(consumes={"text/plain", "application/*"}) 의 경우 HTTP Request에 Content-Type 헤더 값이 "text/plain", "application/*" 모두 매칭이 된다.</p> <p>또한, headers에 대한 입력값으로 Content-Type을 입력한 경우 내부적으로 consumes에 대한 입력값으로 처리된다.</p> <p>예 : @RequestMapping(headers="Content-Type=text/*", consumes="application/*") 의 경우 HTTP Request에 Content-Type 헤더 값이 "text/*", "application/*"인 경우 모두 매칭이 된다.</p>
produces	<p>Spring 3.1에서부터 제공하는 기능으로, HTTP Request의 헤더 값 중에서 'Accept'로 정의된 MediaType 표현식의 배열을 입력값으로 받으며, OR 조건으로 처리된다.</p>

name	Description
	<p>예 : @RequestMapping(produces={"text/html", "application/*"}) 의 경우 HTTP Request에 Accept 헤더 값이 "text/html", "application/*" 모든 경우 매칭이 된다.</p> <p>또한, @RequestMapping 의 header 입력값 중 'Accept'를 입력한 경우 내부적으로 produces에 대한 입력값으로 처리된다.</p> <p>예 : @RequestMapping(headers="Accept=text/*", produces="application/*") 의 경우 HTTP Request에 Accept 헤더 값이 "text/*", "application/*" 인 경우 모두 매칭이 된다.</p>

@RequestMapping은 구현하는 컨트롤러 종류에 따라 아래와 같은 방식으로 사용할 수 있다.

- Form 컨트롤러 구현
- Multi-action 컨트롤러 구현

기존에 SimpleFormController와 같은 컨트롤러 클래스를 상속받아서 컨트롤러를 작성할 때는, 상위 클래스에 정의된 메소드를 override하여 구현하기 때문에 입력 argument 타입과 return 타입이 이미 정해져있다. 이에 반해 **@RequestMapping**을 적용하여 작성하는 핸들러 메소드는 다양한 **argument** 타입과 **return** 타입을 사용할 수 있다.

11.2.2.1.Form 컨트롤러 구현

- 클래스 선언부에 @RequestMapping을 사용하여 처리할 Request URL Mapping
- 메소드에는 @RequestMapping의 'method', 'params'와 같은 상세 속성 정보를 정의하여 Request URL의 Mapping을 세분화

위와 같이 작성하면 기존에 SimpleFormController를 상속받아 작성하였던 폼을 처리하는 컨트롤러를 구현할 수 있다. 다음은 폼 처리 컨트롤러를 작성한 EditMovieController 의 예이다.

```
@Controller
@RequestMapping("/coreMovie.do")
public class EditMovieController {

    @RequestMapping(method = RequestMethod.GET)
    public String createView() {
        // 종락
        return coreViewMovie;
    }

    @RequestMapping(method = RequestMethod.POST)
    public String addMovie(HttpServletRequest request, @ModelAttribute("movie"),
        Movie movie, BindingResult result, SessionStatus status) throws Exception {
        // 종락
        return "redirect:/coreMovieFinder.do";
    }
}
```

11.2.2.2.Multi-action 컨트롤러 구현

@RequestMapping annotation을 사용하여 여러 HTTP Request를 처리할 수 있는 Multi-action 컨트롤러를 구현할 수 있다.

- 메소드에 처리할 Request URL을 Mapping한 @RequestMapping을 정의

다음은 Multi-action 컨트롤러를 구현한 MovieController 의 예이다.

```

@Controller
public class MovieController {

    @RequestMapping("/deleteMovie.do")
    public ModelAndView delete(@RequestParam("movieId") String movieId) {
        // 중략
        return "redirect:/coreMovieFinder.do";
    }

    @RequestMapping("/getMovie.do")
    public String get(@RequestParam("movieId") String movieId, ModelMap model) {
        // 중략
        model.addAttribute(movie);

        return "coreViewMovie";
    }
}

```

11.2.2.3.Supported argument types

@RequestMapping을 사용하여 작성하는 핸들러 메소드는 다음과 같은 타입의 입력 argument를 순서에 관계없이 정의할 수 있다. 단, validation results를 입력 argument로 받을 경우에는 해당 command 객체 바로 다음에 위치해야한다.

- **Servlet API의 Request와 Response 객체**

ServletRequest 또는 HttpServletRequest 등을 메소드 내부에서 직접 사용해야 하는 경우

```

@RequestMapping(params = "param=add")
public String addMovie(HttpServletRequest request,
    Movie movie, BindingResult result, SessionStatus status)
    throws Exception {
    // 중략
    String message = messageSource.getMessage(
        "movie.error.exist", new String[]
    {movie.getMovieId()},
        localeResolver.resolveLocale(request));
}

```

- **Servlet API의 Session**

HttpSession 객체를 메소드 내부에서 사용하는 경우 예 : user 정보와 같은 global session attribute를 사용할 때

```

@RequestMapping("/login.do")
protected ModelAndView handleRequestInternal( HttpSession session,
    @RequestParam("userId") String userId) throws Exception {
    session.setAttribute("userId", userId);
    return new ModelAndView("/index.jsp");
}

```



RequestMappingHandlerAdapter의 'synchronizeOnSession' 속성

Servlet 환경에서 Session 접근은 thread-safe하지 않기 때문에, Session에 저장된 정보에 여러개의 thread가 동시에 접근하여 변경할 가능성이 있는 경우 반드시 RequestMappingHandlerAdapter의 "synchronizeOnSession" 속성을 "true"로 셋팅하도록 한다.

- **java.util.Locale**

현재 request의 locale을 사용할 경우

```
@RequestMapping(params = "param=add")
public String addMovie(Locale locale, Movie movie, BindingResult result,
    SessionStatus status) throws Exception {
    // 중략
    String message = messageSource.getMessage(
        "movie.error.exist", new String[] {movie.getMovieId()},
        locale);
}
```

- **java.io.InputStream 또는 java.io.Reader**

Request의 content를 직접 처리할 경우 (Servlet API가 제공하는 raw InputStream/Reader)

```
@RequestMapping(params = "param=add")
public String addMovie(InputStream is, Movie movie, BindingResult result
    SessionStatus status) throws Exception {
    // 중략
    for(int totalRead = 0; totalRead < totalBytes; totalRead += readBytes) {
        readBytes = is.read(binArray, totalRead, totalBytes - totalRead);
        // 중략
    }
    // 중략
}
```

- **java.io.OutputStream 또는 java.io.Writer**

Response의 content를 직접 처리할 경우 (Servlet API가 제공하는 raw OutputStream/Writer)

```
@RequestMapping(params = "param=add")
public String addMovie(OutputStream os, Movie movie, BindingResult result,
    SessionStatus status) throws
    Exception {
    // 중략
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    byte[] content = outputStream.toByteArray();
    os.write(content);
    os.flush();
    // 중략
}
```

- **@PathVariable annotation이 적용된 argument**

URI template 내의 변수를 핸들러 메소드에서 접근할 경우

@PathVariable에 대한 자세한 사용 방법은 본 매뉴얼 >> Spring REST Plugin
>> URI Template [http://dev.anyframejava.org/docs/anyframe/plugin/optional/springrest/1.1.0/reference/htmlsingle/springrest.html#springrest_restsupport_uritemplate] 참고

```
@RequestMapping(value = "/movies/{movieId}/edit", method = RequestMethod.GET)
public String get(@PathVariable String movieId, Model model)
    throws Exception {
    Movie movie = this.movieService.get(movieId);
    // 중략
    model.addAttribute(movie);
    return "restwebViewMovie";
}
```

- **@RequestParam annotation**이 적용된 **argument**

ServletRequest.getParameter(java.lang.String name)와 같은 역할 수행

```
@RequestMapping(params = "method=remove")
public String remove(@RequestParam("movieId") String movieId)
    throws Exception {
    this.movieService.remove(movieId);
    return "redirect:/coreMovieFinder.do?method=list";
}
```

- **@RequestHeader annotation**이 적용된 **argument**

@RequestHeader를 사용하면 Servlet Request HTTP 헤더 값을 핸들러 메소드에서 사용 가능

```
@RequestMapping("/displayHeaderInfo")
@ResponseBody
public String displayHeaderInfo(@CookieValue("JSESSIONID") String cookie,
    @RequestHeader("Accept-Encoding") String encoding,
    @RequestHeader("Accept") String accept) {
    StringBuilder sf = new StringBuilder();
    sf.append("JSESSIONID : " + cookie);
    sf.append("\n");
    sf.append("Accept-Encoding : " + encoding);
    sf.append("\n");
    sf.append("Accept : " + accept);

    return sf.toString();
}
```

- **@RequestBody annotation**이 적용된 **argument**

@RequestBody를 사용하면 HTTP Request Body를 핸들러 메소드에서 직접 사용 가능

HTTP Request Body가 HttpMessageConverter에 의해서 선언한 메소드 argument 타입으로 변환되어 전달됨

```
@RequestMapping(value = "/movies/add", method = RequestMethod.POST)
@ResponseBody
public String add(@RequestBody Movie movie) throws Exception {
    this.movieService.createMovie(movie);
    return "/movies/" + movie.getMovieId() + "/edit";
}
```

- **HttpEntity<?> 객체**

Servlet request HTTP Header와 Body를 핸들러 메소드에서 접근하기 위해 사용 가능. Request 스트림은 HttpMessageConverter를 통해 entity body로 변환됨.

- **java.util.Map** 또는 **org.springframework.ui.Model** 또는 **org.springframework.ui.ModelMap**

Web View로 데이터를 전달해야 하는 경우 위 타입의 argument를 정의하고, 메소드 내부에서 View로 전달할 데이터를 추가함

```
@RequestMapping("/getMovie.do")
public String getMovie(@RequestParam("movieId") String movieId, Map map) {
    Movie movie = movieService.getMovie(movieId);
    map.put("movie", movie);
    return "/WEB-INF/jsp/annotation/sales/movie/viewMovie.jsp";
}
```

```
@RequestMapping("/getMovie.do")
public String getMovie(@RequestParam("movieId") String movieId, Model model) {
    Movie movie = movieService.getMovie(movieId);
    model.addAttribute("movie", movie);
    return "/WEB-INF/jsp/annotation/sales/movie/viewMovie.jsp";
}
```

```
@RequestMapping("/getMovie.do")
public String getMovie(@RequestParam("movieId") String movieId, ModelMap modelMap) {
    Movie movie = movieService.getMovie(movieId);
    modelMap.addAttribute("movie", movie);
    return "/WEB-INF/jsp/annotation/sales/movie/viewMovie.jsp";
}
```

- **Command 또는 Form 객체**

HTTP Request로 전달된 parameter를 binding한 객체로, 다음 View에서 사용 가능하고 @SessionAttributes를 통해 session에 저장되어 관리될 수 있음. @ModelAttribute annotation을 이용하여 사용자 임의로 이름 부여 가능.

```
@RequestMapping("/addMovie.do")
public String updateMovie(Movie movie, SessionStatus status) throws Exception {
    // 여기서 'movie' 가 Command(또는 Form) 객체이다.
    return "/listMovie.do";
}
```

```
@RequestMapping(params="method=update")
public String update(@ModelAttribute("updatedMovie") Movie movie, SessionStatus status)
throws Exception {
    // 여기서 'updatedMovie' 라는 이름의 'movie' 객체가 Command(/form) 객체이다.
    // 종략
    return "redirect:/coreMovieFinder.do?method=list";
}
```

- **org.springframework.validation.Errors 또는 org.springframework.validation.BindingResult**

바로 이전의 입력파라미터인 Command 또는 Form 객체의 validation 결과 값을 저장하는 객체로, 해당 **Command 또는 Form 객체 바로 다음에 위치해야 함**에 유의

```
@RequestMapping(params = "method=create")
public String create(
    @RequestParam(value="realPosterFile", required=false) MultipartFile posterFile,
    @Valid Movie movie, BindingResult results, SessionStatus status)
throws Exception {
    if (results.hasErrors()) {
        return "coreViewMovie";
    }

    // 종략
    return "redirect:/coreMovieFinder.do?method=list";
}
```

- **org.springframework.web.bind.support.SessionStatus**

Form 처리가 완료되었을 때 status를 처리하기 위해서 argument로 설정. SessionStatus.setComplete()를 호출 하면 컨트롤러 클래스에 @SessionAttributes로 정의된 Model객체를 session에서 지우도록 이벤트 발생

```
@RequestMapping(params = "method=create")
```

```

public String create(
    @RequestParam(value="realPosterFile",required=false) MultipartFile posterFile,
    @Valid Movie movie, BindingResult results, SessionStatus status)
    throws Exception {
    // 종막
    this.movieService.create(movie);
    status.setComplete();
    return "redirect:/coreMovieFinder.do?method=list";
}

```

11.2.2.4.Supported return types

@RequestMapping을 이용한 핸들러 메소드는 다음과 같은 리턴타입을 가질 수 있다.

- **ModelAndView 객체**

View와 Model 정보를 모두 포함한 객체를 리턴하는 경우.

```

@RequestMapping(params="param=addView")
public ModelAndView addMovieView() {
    ModelAndView mnv = new ModelAndView("/WEB-INF/jsp/annotation/sales/movie/
movieForm.jsp");
    mnv.addObject("movie", new Movie());
    return mnv;
}

```

- **Map**

Web View로 전달할 데이터만 리턴하는 경우.

```

@RequestMapping("/movieList.do")
public Map getMovieList() {
    List movieList = movieService.getMovieList();
    ModelMap map = new ModelMap(movieList); //movieList가 "movieList"라는 이름으로 저장됨.
    return map;
}

```

여기서 View에 대한 정보를 명시적으로 리턴하지는 않았지만, 내부적으로 View 이름은 RequestToViewNameTranslator에 의해서 입력된 HTTP Request를 이용하여 생성된다. 예를 들어 DefaultRequestToViewNameTranslator [<http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/web/servlet/view/DefaultRequestToViewNameTranslator.html>] 는 입력된 HTTP Request URI를 변환하여 View 이름을 다음과 같이 생성한다.

```

http://localhost:8080/anyframe-sample/display.do
-> 생성된 View 이름 : 'display'
http://localhost:8080/anyframe-sample/admin/index.do
-> 생성된 View 이름 : 'admin/index'

```

위와 같이 자동으로 생성되는 View 이름에 'jsp/'와 같이 prefix를 붙이거나 '.jsp' 같은 확장자를 덧붙이고자 할 때는 아래와 같이 속성 정의 XML(xxx-servlet.xml)에 추가하면 된다.

```

<bean id="viewNameTranslator"
    class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator">
    <property name="prefix" value="jsp/" />
    <property name="suffix" value=".jsp" />
</bean>

```

- **Model**

Web View로 전달할 데이터만 리턴하는 경우 Model [http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/ui/Model.html]은 Java-5 이상에서 사용할 수 있는 인터페이스이다. 기본적으로 ModelMap과 같은 기능을 제공한다. Model 인터페이스의 구현클래스에는 BindingAwareModelMap [http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/validation/support/BindingAwareModelMap.html] 와 ExtendedModelMap [http://static.springsource.org/spring/docs/4.0.x/javadoc-api/org/springframework/ui/ExtendedModelMap.html] 이 있다. View 이름은 위에서 설명한 바와 같이 RequestToViewNameTranslator에 의해 내부적으로 생성된다.

```
@RequestMapping("/movieList.do")
public Model getMovieList() {
    List movieList = movieService.getMovieList();
    ExtendedModelMap map = new ExtendedModelMap();
    map.addAttribute("movieList", movieList);
    return map;
}
```

- **String**

View 이름만 리턴하는 경우.

```
@RequestMapping(value = {"/addMovie.do", "/updateMovie.do" })
public String updateMovie(Movie movie, SessionStatus status)
    throws Exception {
    // 종략
    return "/listMovie.do";
}
```

- **void**

메소드 내부에서 직접 HTTP Response를 직접 처리하는 경우. 또는 View 이름이 RequestToViewNameTranslator에 의해 내부적으로 생성되는 경우

```
@RequestMapping("/addview.do")
public void addView(HttpServletResponse response) {
    // 종략
    //response 직접 처리
}
```

```
@RequestMapping("/addview.do")
public void addView() {
    // 종략
    // View 이름이 DefaultRequestToViewNameTranslator에 의해서 내부적으로 'addview'로 결정
    됨.
}
```

- **@ResponseBody**

핸들러 메소드의 리턴 객체를 Response HTTP Body로 바로 보내는 경우. HttpResponseMessage를 통해서 리턴 객체가 변환되어 Response로 전달됨.

```
@RequestMapping(value = "/welcome", method = RequestMethod.GET)
@ResponseBody
public String welcome() {
    return "welcome!";
}
```

- **HttpEntity<?> 또는 ResponseEntity<?>**

Response HTTP의 Body와 Header를 핸들러 메소드에서 접근하기 위해 사용 가능. HttpEntity나 ResponseEntity의 Body는 HttpMessageConverter를 통해 response 스트림으로 변환됨.

```
@RequestMapping(value = "/welcome", method = RequestMethod.GET)
@ResponseBody
public String welcome() {
    return "welcome!";
}
```

11.2.3. @RequestParam

@RequestParam annotation은 HTTP Request parameter를 컨트롤러 메소드의 argument로 바인딩하는데 사용되며 ServletRequest.getParameter(java.lang.String name) [[http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/ServletRequest.html#getParameter\(java.lang.String\)](http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/ServletRequest.html#getParameter(java.lang.String))] 와 같은 역할을 한다. 다음은 @RequestParam annotation의 사용 예이다.

```
@RequestMapping("/updateMovie.do")
public String updateMovie(@RequestParam("movieId") String movieId,
    @RequestParam("sellAmount") int sellAmount, @RequestParam("realImageFile")
    MultipartFile picturefile) {
    // 중략
    return "/listMovie.do";
}
```

@RequestParam을 적용한 파라미터는 반드시 HTTP Request에 존재해야 한다. 그렇지 않은 경우 다음과 같이 org.springframework.web.bind.MissingServletRequestParameterException이 발생한다.

```
org.springframework.web.bind.MissingServletRequestParameterException:
Required java.lang.String parameter 'movieId' is not present
```

그러나 아래와 같이 @RequestParam의 required 속성을 false로 설정할 경우 HTTP Request에 파라미터가 존재하지 않아도 Exception이 발생하지 않는다.

```
@RequestMapping("/deleteMovie.do")
public String deleteMovie(@RequestParam(value="movieId", required="false") String movieId){
    // 중략
}
```

또한 defaultValue 속성을 이용하여 해당 파라미터가 존재하지 않을 경우 사용할 디폴트 값을 정의할 수 있다.

```
@RequestMapping("/movies.do")
public String findMovies(@RequestParam(value="pageIndex", defaultValue = "1") int pageIndex,
    Movies movies, BindingResult result, Model model) {
    // 중략
}
```

11.2.4. @RequestBody

@RequestBody annotation은 HTTP Request Body를 컨트롤러 메소드의 argument로 바인딩하는데 사용된다. 다음은 @RequestBody annotation의 사용 예이다.

```
@RequestMapping(value = "/movies/add", method = RequestMethod.POST)
@ResponseBody
public String add(@RequestBody Movie movie) throws Exception {
```



```
// 종락
}
```

Request Body의 내용을 메소드의 argument 객체로 전달하기 위해서는 `HttpMessageConverter`에 의해서 변환이 이루어져야만 한다. `HttpMessageConverter`는 HTTP Request body와 객체간, 그리고 객체와 HTTP Response body간의 변환을 담당한다. Spring 3 부터 `AnnotationMethodHandlerAdapter`가 `@RequestBody`를 지원하고, 다음의 `HttpMessageConverter` 들을 디폴트로 등록하도록 기능이 확장되었다.

- `ByteArrayHttpMessageConverter` : byte 배열로 변환
- `StringHttpMessageConverter` : String으로 변환
- `FormHttpMessageConverter` : Form 데이터와 `MultiValueMap<String, String>` 간의 변환
- `SourceHttpMessageConverter` : `javax.xml.transform.Source`로 변환
- `MarshallingHttpMessageConverter` : `org.springframework.xml` 패키지에서 제공하는 `Marshaller`와 `Unmarshaller`를 사용하여 객체와 XML간 변환
- `MappingJacksonHttpMessageConverter` : Jackson 라이브러리의 `ObjectMapper`를 사용해서 객체와 JSON 간의 변환

위와 같은 `MessageConverter`들이 어플리케이션에서 사용되려면 `AnnotationMethodHandlerAdapter`에 설정되어 있어야한다. `AnnotationMethodHandlerAdapter`에 "**messageConverters**" 속성을 이용하여 설정할 수도 있지만, 앞서 언급했던 `<mvc:annotation-driven />`만 정의하면 디폴트로 자동으로 등록해준다. `MessageConverter`에 대한 더 자세한 내용은 본 매뉴얼 >> Spring REST Plugin >> HTTP Message Conversion [http://dev.anyframejava.org/docs/anyframe/plugin/optional/springrest/1.1.0/reference/htmlsingle/springrest.html#springrest_restsupport_httpmessageconversion]을 참고하기 바란다.

11.2.5.@ResponseBody

`@ResponseBody` annotation은 핸들러 메소드가 리턴 값을 HTTP Response를 통해서 바로 전달할 경우에 사용할 수 있다. `@ResponseBody`가 적용되면 Model과 View를 리턴하여 `ViewResolver`를 통해 View를 찾는 등의 과정들은 거치지 않게 된다.

다음은 `@ResponseBody` annotation의 사용 예이다.

```
@RequestMapping(value = "/welcome", method = RequestMethod.GET)
@ResponseBody
public String welcome() {
    return "welcome!";
}
```

위에서 설명했던 `@RequestBody`에서와 같이, 핸들러 메소드의 리턴값은 `HttpMessageConverter`를 통해 HTTP Response Body로 변환된다.

11.2.6.@RestController

`@RestController` Annotation은 `@Controller` Annotation과 `@ResponseBody` Annotation을 합친 동작을 수행한다. `@Controller`와 마찬가지로 Stereotype annotation으로 component scan 이후 bean으로 등록해서 사용할 수 있다. 또한 `@RestController`로 선언한 메소드의 모든 리턴 값을 HTTP Response를 통해서 바로 전달하게 되며, 따라서 `ViewResolver`를 통해 View를 찾는 등의 과정을 거치지 않게 된다.

다음은 `@RestController`의 사용 예이다.

```
@RestController
@RequestMapping("/restmovies")
public class RestMovieController {
```

```

@Inject
@Named("springrestMovieService")
private MovieService movieService;

/**
 * Generate response
 *
 * @param movieId
 * @return return Movie object, Response Status : 200 OK
 * @throws Exception
 */
@RequestMapping(value = "/{movieId}", method = RequestMethod.GET)
public Movie get(@PathVariable String movieId) throws Exception {
    Movie movie = this.movieService.get(movieId);

    if (movie == null) {
        throw new NotFoundException("Resource Not Found with movieId "
            + movieId);
    }

    return movie;
}

```

위에서 설명했던 `@ResponseBody`에서와 같이, 핸들러 메소드의 리턴값은 `HttpMessageConverter`를 통해 HTTP Response Body로 변환된다.

11.2.7.HttpEntity<?>

`HttpEntity`는 `@RequestBody`/`@ResponseBody` 같이 Request/Response Body 메시지를 처리할 수 있을 뿐만 아니라, HTTP Header 값도 함께 다룰 수 있다. 일반적으로 `RestTemplate`을 사용한 REST 클라이언트를 구현할 때, 편리하게 사용될 수 있다.

```

@RequestMapping("/handle")
public HttpEntity<String> handle() {
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("MyResponseHeader", "MyValue");
    return new ResponseEntity<String>("Hello World", responseHeaders);
}

```

`HttpEntity`를 사용할 경우에도 역시 Request/Response Body 변환을 위해 `HttpMessageConverter`가 사용된다.

11.2.8.@ModelAttribute

`@ModelAttribute`는 컨트롤러에서 다음과 같이 두 가지 방법으로 사용할 수 있다.

- 메소드 자체에 정의

입력 폼 페이지에서 출력해 줄 reference data를 전달하고자 할 때. 기존 `SimpleFormController` [<http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/servlet/mvc/SimpleFormController.html>]의 `referenceData()` 메소드와 같은 역할

- 메소드의 입력 argument에 정의

메소드의 argument로 입력된 Command 객체에 이름을 부여하고자 할 때.

다음은 위에서 설명한 두가지 방법으로 `@ModelAttribute`를 사용한 예이다.

```

@Controller
@RequestMapping("/movie.do")
public class MovieController {
    // 종략
    // 메소드 자체에 정의
    @ModelAttribute("genreList")
    public Collection<Genre> populateGenreList() throws Exception {
        return this.genreService.getDropDownGenreList();
    }

    // 메소드의 입력 argument에 정의
    @RequestMapping(params="method=add")
    public String add(@ModelAttribute("updatedMovie") Movie movie
                     , BindingResult result, SessionStatus status) throws
    Exception {
        // 종략
    }
}

```

11.2.9. @SessionAttributes

@SessionAttributes는 Session에 저장하여 관리할 Model Attribute를 정의할 때 사용한다. Session에 저장하고자 하는 Model Attribute의 이름이나 타입을 @SessionAttributes의 속성에 정의해준다.

다음은 @SessionAttributes를 사용하여 Session에 저장하여 관리할 Model을 이름으로 정의한 예이다. 타입으로 정의할 경우 'types'라는 속성을 이용한다.

```

@Controller
@RequestMapping("/movie.do")
@SessionAttributes(value={"movie", "genre"})
public class MovieController {
    // 종략
}

```

11.2.10. @CookieValue

HTTP Cookie에 저장된 값을 핸들러 메소드에서 사용할 수 있도록 해주는 Annotation이다.

다음은 @CookieValue 사용하여 Cookie 값을 가져와 출력해보는 코드이다.

```

@RequestMapping("/displayHeaderInfo")
@ResponseBody
public String displayHeaderInfo(@CookieValue("JSESSIONID") String cookie,
    @RequestHeader("Accept-Encoding") String encoding,
    @RequestHeader("Accept") String accept) {
    StringBuilder sf = new StringBuilder();
    sf.append("JSESSIONID : " + cookie);
    sf.append("\n");
    sf.append("Accept-Encoding : " + encoding);
    sf.append("\n");
    sf.append("Accept : " + accept);

    return sf.toString();
}

```

11.2.11. @RequestHeader

HTTP Header에 저장된 값을 핸들러 메소드에서 사용할 수 있도록 해주는 Annotation이다.

위 @CookieValue 예제 코드에서 @RequestHeader가 사용된 모습을 확인할 수 있다.

11.3.Double Form Submission 방지

입력 폼 페이지에서 사용자가 새로 고침 버튼을 클릭하거나, 폼을 Submit하는 버튼을 여러번 클릭할 경우 같은 입력 폼 정보가 서버로 여러번 등록되는 문제가 발생할 수 있다. 이 장에서는 이러한 Double Form Submission을 어떻게 방지할 수 있는지를 알아보도록 하자.

Double Form Submission 방지는 다음과 같은 원리로 구현된다.

- 반드시 **RequestMappingHandlerAdapter**의 **synchronizeOnSession** 속성을 **true**로 설정

```
<bean id="requestMappingHandlerAdapter"
      class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="synchronizeOnSession" value="true" />
</bean>
```

- **Double submission**을 방지하고자 하는 **Form** 객체를 **model**로 저장

다음 예제와 같이 ModelAndView, ModelMap 등을 이용하여 저장한다.

```
@RequestMapping(params = "param=addView")
public ModelAndView addMovieView() {
    ModelAndView mnv =
        new ModelAndView("/WEB-INF/jsp/annotation/sales/movie/movieForm.jsp");
    mnv.addObject("movie", new Movie());
    return mnv;
}
```

- 저장한 **model**을 **@SessionAttributes**로 정의

다음 예제와 같이 컨트롤러 클래스 선언부에 @SessionAttributes("movie")로 정의한다.

```
@Controller
@RequestMapping("/movie.do")
@SessionAttributes("movie")
public class EditMovieController {
    // 중략
}
```

- 컨트롤러 메소드에서 폼 처리 완료 후 **Session status** 변경

```
@RequestMapping(params = "param=add")
public String addMovie(HttpServletRequest request, Movie movie, BindingResult result
    , SessionStatus status) throws Exception {
    movieService.addMovie(movie);
    status.setComplete();
    return "/listMovie.do";
}
```

- **status.setComplete()**는 **session**에서 저장된 **model**을 삭제하는 이벤트 발생
- 따라서, 이후에 다시 **submit** 요청이 온 경우 **session**에 저장된 **model**이 삭제되었기 때문에 아래와 같이 **org.springframework.web.HttpSessionRequiredException**발생

```
org.springframework.web.HttpSessionRequiredException:
    Session attribute 'dept' required - not found in session
```

12.View

Spring MVC는 JSP에서 보다 쉽게 데이터를 출력할 수 있도록 Tag Library를 제공하며 여러 View 기술 (Velocity, Freemarker, Tiles 등)과의 연계 방법을 제시한다.

12.1.Tag library

Spring MVC에서는 입력 폼 구현을 보다 쉽게 구현하기 위해 Spring Form Tag를 제공한다. 이는 태그에서 command 객체, controller 참조 데이터로의 접근이 가능하다. Spring Form tag의 사용 방법은 매우 간단하며 예제를 중심으로 각 tag에 대한 내용을 살펴본다.

12.1.1.configuration

Spring Form Tag를 사용하기 위해서는 spring-form.tld 파일이 필요하고 이는 spring-webmvc-x.x.x.jar 파일에 포함되어 있다. 이 폼 태그를 사용하기 위해서는 JSP 페이지에 taglib을 추가해줘야한다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

12.1.2.form

<form>은 데이터 바인딩을 위해 태그 안에 바인딩 path를 지정해 줄 수 있다. path에 해당되는 값은 도메인 모델의 Bean 객체를 의미한다. 사용예는 다음과 같다.

```
<form:form commandName="user">
    userId : <form:input path="userId"/>
</form:form>
```

또한 Spring Form Tag를 이용하기 위해서는 각각의 입력 path값에 매칭될 트랜스퍼 오브젝트를 지정해 줘야하는데 <form>안에 commandName 속성으로 다음과 같이 지정해 줄 수 있다.

```
<% request.setAttribute("user", sample.services.UserVO())>
```

이러한 commandName의 기본값은 "command"이며 input값들과 매칭될 트랜스퍼 오브젝트를 request 값으로 셋팅해줘야한다. 이 값은 SimpleFormController를 사용할 경우 formBackingObject()메소드에서 지정해 줄 수도있다.

```
protected Object formBackingObject(HttpServletRequest request)
    throws Exception {
    UserVO vo=new UserVO();
    request.setAttribute("user",vo);
    return new UserVO();
}
```

12.1.3.input

HTML의 <input>의 value가 text인 것을 기본 value로 갖는다. 이 태그의 예는 위의 <form> 예에서 볼 수 있다.

12.1.4.checkbox

다음은 <checkbox>의 예이다. 마찬가지로 path에 트랜스퍼 오브젝트의 bean name을 매핑시켜주고 label속성을 이용하면 jsp페이지로 보여질 이름을 설정할 수 있다.

```
<form:checkbox path="hobby" value="listeningMusic" label="음악감상"/>
<form:checkbox path="hobby" value="study" label="공부"/>
```

※ 위 코드는 아래와 같은 화면을 출력한다.

hobby : ☐ 음악감상 ☐ 공부

12.1.5.checkboxes

위의 <checkbox>는 각각의 항목에 대해 작성해줘야 하지만 <checkboxes>를 사용하면 items속성을 이용해서 한줄로 나타내줄 수 있다. 이러한 items에 들어갈 값은 컨트롤러의 formBackingObject()메소드에서 Array, List, Map형태의 것들로 넘겨 줄 수 있다. Map의 key와 value쌍으로 넘겨줄 경우 key는 태그의 value값이 되고 value는 label명이 된다. (단, Array나 List로 넘길 경우 label은 value와 같은 값을 가지게 된다.) 다음은 그 예이다.

```
protected Object formBackingObject(HttpServletRequest request) throws Exception {
    UserVO vo = new UserVO();
    Map<String, String> interest = new HashMap<String, String>();
    interest.put("reading", "독서");
    interest.put("listeningMusic", "음악감상");
    interest.put("study", "공부");
    request.setAttribute("interest", interest);
    request.setAttribute("user", vo);
    return new UserVO();
}
```

```
<tr>
    <td>hobby :</td>
    <td><form:checkboxes path="hobby" items="${interest}" /></td>
</tr>
```

※ 위 코드는 아래와 같은 화면을 출력한다.

hobby : ☐ 공부 ☐ 독서 ☐ 음악감상

12.1.6.radiobutton

다음은 <radiobutton>의 예이다. <radiobutton> 또한 label 속성을 이용하여 label명을 설정해 줄 수 있다.

```
<tr>
    <td>Gender:</td>
    <td>Male: <form:radiobutton path="gender" value="M" label="남자"/> <br/>
        Female: <form:radiobutton path="gender" value="F" label="여자"/> </td>
</tr>
```

sex : ☐ 남자 ☐ 여자

12.1.7.radiobuttons

다음은 <radiobuttons>의 예이다. items 속성의 사용방법은 위의 <checkboxes>와 동일하다.

```
<tr>
    <td>Gender:</td>
    <td><form:radiobuttons path="gender" items="${genderOptions}" /></td>
</tr>
```

12.1.8.password

다음은 <password>의 예이다.

```
<tr>
  <td>password :</td>
  <td><form:password path="password" /></td>
</tr>
```

※ 위 코드는 아래와 같은 화면을 출력한다.

password :

12.1.9.select


<select>도 위의 <checkboxes>나 <radiobuttons>처럼 items 속성을 이용하여 formBackingObject에서 넘겨주는 값으로 자동 매핑 시켜줄 수 있다.

```
protected Object formBackingObject(HttpServletRequest request)
    throws Exception {
    UserVO vo = new UserVO();
    Map<String, String> address = new HashMap<String, String>();
    address.put("seoul", "서울");
    address.put("daegu", "대구");
    address.put("busan", "부산");
    request.setAttribute("address", address);
    request.setAttribute("user", vo);

    return new UserVO();
}
```

```
<tr>
  <td>주소</td>
  <td><form:select path="address" items="${address}" /></td>
</tr>
```

※ 위 코드는 아래와 같은 화면을 출력한다.

주소 

일반적인 <option>와 함께 아래와 같이 사용할 수도 있다.

12.1.10.option

다음은 <option>의 사용 예이다.

```
<tr>
  <td>주소</td>
  <td><form:select path="address">
    <form:option value="seoul" label="서울" />
    <form:option value="daegu" label="대구" />
    <form:option value="busan" label="부산" />
  </form:select></td>
</tr>
```

12.1.11.options

다음은 <options>의 사용예이다.

```
<tr>
  <td>주소</td>
  <td><form:select path="address">
    <form:options items="${address}" />
  </form:select></td>
</tr>
```

12.1.12.textarea

다음은 <textarea>의 사용 예이다.

```
<td>Note :</td>
<td><form:textarea path="comment" rows="3" cols="20"></form:textarea></td>
```

Note :

12.1.13.hidden

다음은 <hidden>의 사용 예이다.

```
<form:hidden path="userId" />
```

12.1.14.errors

Spring MVC는 validator에서 얻어진 메시지를 JSP페이지에서 쉽게 출력할 수 있도록 Spring Form 태그의 <form:errors>를 제공한다. 이는 생성한 validator를 통해 입력값의 유효성 체크 후 에러 메시지를 출력해주는데 자세한 사항은 본 매뉴얼 Spring MVC >> Validation >> Spring Validator의 <form:errors> 태그 사용을 참고한다.

12.1.15.sample

12.1.15.1.입력 화면

다음은 영화 정보 입력 화면 작성 예인 form.jsp 파일의 일부이다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<form:form modelAttribute="movie" name="movieForm" method="post">
  <c:if test="${not empty movie.movieId}">
    <form:hidden path="movieId" />
  </c:if>
  <table summary="This table shows detail information about genre, title, director, actors,
runtime, release date, ticket price of the movie">
    <caption>Detail information</caption>
    <colgroup>
      <col style="width:20%;" />
      <col style="width:80%;" />
    </colgroup>
    <tbody>
```



```

<tr>
  <th><label for="title"><spring:message code="movie.title" />&nbsp;</label></th>
  <td><form:input path="title" cssClass="w_normal" /><form:errors path="title"
cssClass="errors" /></td>
</tr>
<tr>
  <th><label for="director"><spring:message code="movie.director" />&nbsp;</label></th>
  <td><form:input path="director" cssClass="w_normal" /><form:errors path="director"
cssClass="errors" /></td>
</tr>
<tr>
  <th><label for="genre"><spring:message code="movie.genre" />&nbsp;</label></th>
  <td>
    <!-- items 속성을 사용하여 컨트롤러의 populateGenreList() 에서 넘겨준
    Map 형태의 객체를 받아서 출력해준다. -->
    <form:select id="genre" path="genre.genreId">
      <form:options items="{genreList}" itemValue="genreId" itemLabel="name"/>
    </form:select>
  </td>
</tr>
<tr>
  <th><label for="actors"><spring:message code="movie.actors" />&nbsp;</label></th>
  <td><form:input path="actors" cssClass="w_normal" /><form:errors path="actors"
cssClass="errors" /></td>
</tr>
<tr>
  <th><label for="runtime"><spring:message code="movie.runtime" /></label></th>
  <td><form:input path="runtime" cssClass="w_time" />min.<form:errors path="runtime"
cssClass="errors" /></td>
</tr>
<tr>
  <th><label for="releaseDate"><spring:message code="movie.releaseDate" /></label></th>
  <td>
    <form:input path="releaseDate" cssClass="w_date" maxLength="10" />
    <a class="underline_none" href="javascript:popupCalendar(document.movieForm.releaseDate,
'yyyy-mm-dd');">
      
    </a>
    <form:errors path="releaseDate" cssClass="errors" /></td>
</tr>
<tr>
  <th><label for="ticketPrice"><spring:message code="movie.ticketPrice" /></label></th>
  <td><form:input path="ticketPrice" cssClass="w_price" /><form:errors path="ticketPrice"
cssClass="errors" /></td>
</tr>
<tr>
  <th><label for="nowPlaying"><spring:message code="movie.nowPlaying" /></label></th>
  <td><span class="float_left"><spring:message code="movie.isNowPlaying" /></span>
    <span class="float_left margin_left5"><form:checkbox id="nowPlaying" path="nowPlaying"
value="Y" /></span>
    <input type="hidden" name="!nowPlaying" value="N" /></td>
</tr>
</tbody>
</table>
</form:form>

```

12.1.15.2.Controller 클래스

다음은 Form에서 사용할 객체를 셋팅해주는 MovieController 파일의 populateGenreList()메소드와 요청 처리 결과를 모델 객체에 셋팅해서 view로 넘겨주는 create()메소드의 일부이다.

```

@Controller("movieController")
@RequestMapping("/movie.do")
public class MovieController {
    ... 중략...

    // genreList select box에서 사용할 데이터를 view로 전달
    @ModelAttribute("genreList")
    public Collection<Genre>populateGenreList() throws Exception {
        return this.genreService.getList();
    }

    @RequestMapping(params = "method=create")
    public String create(
        @Valid Movie movie, BindingResult results, SessionStatus status, Model model,
        HttpSession session) throws Exception {

        if (results.hasErrors())
            return "core/moviefinder/movie/form";

        // call business service
        this.movieService.create(movie);
        status.setComplete();

        // view에 "movie" 라는 객체를 넘겨준다.
        model.addAttribute("movie", movie);

        return "moviefinder/movie/get";
    }
}

```

12.1.15.3. 출력 화면

다음은 EL문을 사용한 데이터 출력을 작성한 get.jsp 파일의 일부이다.

```

<tr><td>Title : </td><td>${movie.title}</td></tr>
<tr><td>Director : </td><td>${movie.director}</td></tr>
<tr><td>actors : </td><td>${movie.actors}</td></tr>
<tr><td>runtime : </td><td>${movie.runtime}</td></tr>
... 중략

```

위의 JSP 코드처럼 Expression Language(JSP 2.0에서 지원)를 사용하여 Controller에서 넘겨준 "movie"라는 이름의 모델 객체의 값을 출력할 수 있다.

12.2. Tiles Integration

Tiles Plugin 매뉴얼 [<http://dev.anyframejava.org/docs/anyframe/plugin/optional/tiles/1.1.0/reference/htmlsingle/tiles.html>]을 참조한다.

13.Validation

Spring에서는 사용자가 입력한 값에 대한 유효성을 체크하기 위해 Spring Validator 또는 JSR-303 Validator를 사용할 수 있도록 지원하고 있다.

13.1.Spring Validator

Spring MVC에서는 Spring Validator를 이용하여 입력 필드의 값에 대해 Validation Check를 수행하고 Errors 객체를 통해 에러 메시지를 출력해 줄 수 있도록 지원한다. 또한 Errors 객체에 담겨진 에러 메시지는 jsp 페이지에서 form:errors 태그를 통해 출력될 수 있다.

13.1.1.Validator 생성

- ValidatorUtils 사용

필수 입력 필드에 대해 Validation Check를 수행하고 에러 메시지를 출력할 수 있도록 지원한다. 이것은 ValidatorUtils를 사용하여 간단히 구현할 수 있다. 다음은 Validator 인터페이스를 구현한 UserValidator.java 클래스의 일부이다.

```
public class UserValidator implements Validator {
    public boolean supports(Class clazz) {
        return UserVO.class.isAssignableFrom(clazz);
    }

    public void validate(Object object, Errors errors) {
        // validationUtils를 이용하여 입력값이 비었는지 체크
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "userName",
            "required", new Object[] { "userName" }, "Enter your name");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password",
            "required", new Object[] { "password" }, "Enter your password");
    }
}
```

- Errors 사용

Validation Check 결과 발생한 Error를 Errors 객체를 사용하여 저장함으로써 해당 필드에 대해 정의된 에러 메시지를 출력할 수 있도록 지원하며 그 예는 다음과 같다.

```
public class UserValidator implements Validator {
    public boolean supports(Class clazz) {
        return HelloVO.class.isAssignableFrom(clazz);
    }

    public void validate(Object object, Errors errors) {

        HelloVO helloVO = (HelloVO) object;
        if (helloVO.getPassword().length() < 6)
            errors.rejectValue("password", "error.password.tooshort");

        if (!helloVO.getPassword().equals(helloVO.getConfirmPassword()))
            errors.rejectValue("confirmPassword", "error.confirm");
    }
}
```

Validation Error가 있는 경우 메시지 리소스 파일에 미리 정의된 error.password.tooshort, error.confirm 등의 메시지가 출력될 것이다.

13.1.2.Validator 활용

생성한 Validator를 활용하기 위해서는 해당 Validator를 Inject하여 사용하거나 Controller 클래스 내에 @InitBinder 메소드를 정의하고 해당 메소드의 입력 인자로 전달된 Binder에 해당 Validator를 셋팅하여 활용할 수 있다.

13.1.3.<form:errors> 태그 사용

Validation Error를 JSP 페이지에서 쉽게 출력하기 위해 Spring MVC에서 제공하는 form 태그 중 <form:errors> 태그를 사용할 수 있다. 이 태그를 사용하기 위해서는 다음과 같은 절차를 따르도록 한다.

- 태그 라이브러리 등록

Spring form 태그 라이브러리를 사용하기 위해서는 spring-form.tld 파일이 필요하며 이는 spring-webmvc-x.x.x.jar 파일에 포함되어 있다. 이 form 태그를 사용하기 위해서는 JSP 페이지에 다음과 같이 taglib 정의가 추가되어야 한다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

- <form:form> 태그 사용

form 태그를 사용하려면 commandName 속성을 지정해야 하는데 이 이름은 JSP 페이지에서 사용되는 commandName과 일치해야 하며 commandClass와 같은 타입의 객체이어야 한다. commandName에 특정 이름을 부여하지 않으면 기본 값은 command로 셋팅된다. form 태그는 여러가지 폼 입력 태그들을 갖는다. 그 중, Validation Error 표현을 위한 태그는 <form:errors>이며 이 태그는 속성으로 path를 가진다. path 값으로 "*" 값을 주게 되면 commandClass가 가지는 모든 속성에 대한 Error 메시지를 출력하게 된다. 다음은 <form:errors> 태그가 정의되어 있는 getUser.jsp 파일의 일부이다.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<tr>
  <td> Name :</td>
  <td><form:input path="userName" />(required)</td>
  <td><form:errors path="userName" /></td>
</tr>
<tr>
  <td>password :</td>
  <td><form:password path="password" />(required, 6자 이상입력)</td>
  <td><form:errors path="password" /></td>
</tr>
```

13.2.Spring 3 Validation

Spring 3 이후부터는 Bean Validation에 대한 표준을 정의한 JSR-303 Spec.을 지원하고 있다. Validation은 선언적인 형태와 프로그램적인 형태로 구분할 수 있으며 Hibernate Validator와 같은 JSR-303 Spec.을 구현한 구현체를 연계하여 처리된다.

13.2.1.JSR-303 (Bean Validation) Basic

JSR-303은 Bean Validation을 위한 표준을 정의하고 있으며 특정 어플리케이션을 구성하는 도메인 클래스에 대해 JSR-303 Annotation을 활용하여 Validation Constraints를 부여하게 되면 런타임시에 이를 기준으로 Validation Check가 이루어지게 된다. 다음은 JSR-303 Spec.에서 제시한 Annotation 목록이다.

Annotation	Supported Type	Description
@AssertFalse	boolean, Boolean	해당 속성의 값이 false 인지 체크한다.

Annotation	Supported Type	Description
@AssertTrue	boolean, Boolean	해당 속성의 값이 true인지 체크한다.
@DecimalMax	BigDecimal, BigInteger, String, byte, short, int, long and primitive type에 대한 wrappers	해당 속성이 가질 수 있는 최대값을 체크한다.
@DecimalMin	BigDecimal, BigInteger, String, byte, short, int, long and primitive type에 대한 wrappers	해당 속성이 가질 수 있는 최소값을 체크한다.
@Digits	BigDecimal, BigInteger, String, byte, short, int, long and primitive type에 대한 wrappers	해당 속성이 가질 수 있는 정수부의 자리수와 소수부의 자리수를 체크한다.
@Future	java.util.Date, java.util.Calendar	해당 속성의 값이 현재일 이후인지 체크한다.
@Max	BigDecimal, BigInteger, String, byte, short, int, long and primitive type에 대한 wrappers	해당 속성이 가질 수 있는 최대 Length를 체크한다.
@Min	BigDecimal, BigInteger, String, byte, short, int, long and primitive type에 대한 wrappers	해당 속성이 가질 수 있는 최소 Length를 체크한다.
@NotNull	any type	해당 속성의 값이 Null이 아닌지 체크한다.
@Null	any type	해당 속성의 값이 Null인지 체크한다.
@Past	java.util.Date, java.util.Calendar	해당 속성의 값이 현재일 이전인지 체크한다.
@Pattern	String	해당 속성의 값이 정의된 Regular Expression에 부합하는지 체크한다. Regular Expression은 Java Regular Expression Convention(see java.util.regex.Pattern)에 맞게 정의해야 한다.
@Size	String, Collection, Map and arrays	해당 속성이 가질 수 있는 최대, 최소 Length를 체크한다.
@Valid	any non primitive type	해당 객체에 대해 Validation Check가 이루어진다.

이 외에도 JSR-303 구현체별로 Validation Constraint 정의를 위한 Custom Annotation을 추가로 제공할 수도 있다.

다음은 Core Plugin 설치로 추가된 도메인 클래스 ~/domain/Movie.java의 일부로써 앞서 언급한 JSR-303 Annotation을 활용하여 Validation Constraint를 정의하고 있다. 예를 들어, title 속성은 Null 값을 가질 수 없으며 최소 1자리, 최대 50자리까지만 허용하며 runtime 속성값은 최대 180을 초과할 수 없고 정수부 3자리 소수부는 0자리를 허용하고 있음을 알 수 있다.

```
public class Movie implements Serializable {
    private static final long serialVersionUID = 1L;
```

```

private String movieId;

@NotNull
@Size(min = 1, max = 50)
private String title = "";

@NotNull
@Size(min = 1, max = 50)
private String director;

private Genre genre;

@NotNull
@Size(min = 5, max = 100)
private String actors;

@DecimalMax(value = "180")
@Digits(integer=3, fraction=0)
private int runtime;

@DateTimeFormat(iso = ISO.DATE)
@Future
private Date releaseDate;

@NumberFormat(pattern = "#,###")
@Digits(integer=5, fraction=0)
private int ticketPrice;

private String posterFile;

private String nowPlaying = "Y";

// getter, setter ...
}

```

13.2.2.JSR-303 (Bean Validation) Optional

JSR-303 Spec.을 준수하는 모든 Annotation은 Annotation별 속성 외에 payload, groups, message라는 속성을 공통적으로 가진다. 각 속성이 가지는 의미에 대해 살펴보도록 하자.

- payload (Programmatic Validating의 경우 활용 가능) : 사용된 Validation Constraint와 관련된 메타 정보를 정의하는데 사용된다. 특정 Constraint에 대해 payload 속성의 값으로 심각도를 정의해두면 Validation Error가 발생하였을 경우 심각도 정보를 추출할 수 있게 된다. 다음은 payload 정보가 추가 정의된 도메인 클래스의 일부로 title, director의 Validation Constraint에 대해 Severity라는 클래스의 Error와 Warning 클래스로써 payload 값을 부여하고 있음을 알 수 있다.

```

public class Movie implements Serializable {

    private static final long serialVersionUID = 1L;

    private String movieId;

    @NotNull(payload = Severity.Error.class)
    @Size(min = 1, max = 50, payload = Severity.Warning.class)
    private String title = "";

    @NotNull(payload = Severity.Error.class)
    @Size(min = 1, max = 50, payload = Severity.Warning.class)
    private String director;
}

```

```
// ...
}
```

다음은 위에서 언급한 Severity 클래스의 모습이다. 내부에 Warning, Error라는 클래스 정의를 포함하고 있으며 이들 각각은 javax.validation.Payload를 상속받고 있음에 유의해야 한다.

```
public class Severity {
    public static interface Warning extends Payload {
    };

    public static interface Error extends Payload {
    };
}
```

위와 같이 코드가 구성된 경우 Validation Error를 담고 있는 ConstraintViolation 객체의 getConstraintDescriptor().getPayload() 메소드를 호출함으로써 Payload 정보를 추출할 수 있다.

```
Set<ConstraintViolation<Movie>> constraintViolations = validator.validate(movie);
System.out.println("the number of constraint violation is " +
    constraintViolations.size());

Iterator<ConstraintViolation<Movie>> iterator = constraintViolations.iterator();

while (iterator.hasNext()) {
    ConstraintViolation<Movie> constraintViolation = iterator.next();
    Set payloads = constraintViolation.getConstraintDescriptor().getPayload();
    // ...
}
```

위에서 언급한 payload 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-validation- payload를 통해 다운로드받을 수 있다.

- groups (Programmatic Validating의 경우 활용 가능) : 사용된 Validation Constraint의 그룹 정보를 정의하는데 사용된다. 일부 Constraint에 대해 동일한 그룹을 부여하게 되면 특정 그룹에 대해서만 Validation 작업을 수행할 수 있게 된다. 예를 들어, 특정 도메인 객체가 생성되는 시점의 Validation Check 대상 속성들과 해당 도메인 객체가 수정되는 시점의 Validation Check 대상 속성들이 다를 수 있기 때문에 이들에 대해 그룹을 부여하고 그룹별로 Validation을 수행하고자 하는 경우 활용할 수 있다. 다음은 groups 정보가 추가 정의된 도메인 클래스의 일부로 title, director, actors에 대해서는 Draft, Playing이라는 그룹을 부여하고 runtime, releaseDate, ticketPrice에 대해서는 Playing이라는 그룹만 부여하고 있음을 알 수 있다.

```
public class Movie implements Serializable {
    private static final long serialVersionUID = 1L;

    private String movieId;

    @NotNull(groups = { Draft.class, Playing.class })
    @Size(min = 1, max = 50, groups = { Draft.class, Playing.class })
    private String title = "";

    @NotNull(groups = { Draft.class, Playing.class })
    @Size(min = 1, max = 50, groups = { Draft.class, Playing.class })
    private String director;

    @NotNull(groups = { Draft.class, Playing.class })
    @Size(min = 5, max = 100, groups = { Draft.class, Playing.class })
    private String actors;
```

```

    @DecimalMax(value = "180", groups = Playing.class)
    @Digits(integer = 3, fraction = 0, groups = Playing.class)
    private int runtime;

    @Future(groups = Playing.class)
    private Date releaseDate;

    @Digits(integer = 5, fraction = 0, groups = Playing.class)
    private int ticketPrice;

    // ...
}

```

즉, 영화가 등록될 당시(Draft Group)에는 title, director, actors에 대해서만 Validation Check가 이루어지고 영화 상영이 결정된 이후(Playing Group)부터는 runtime, releaseDate, ticketPrice에 대해서도 추가적으로 Validation Check가 이루어질 수 있도록 하기 위함이다.

다음은 위에서 groups 정의시 활용한 Draft.java 클래스의 모습이다. group 클래스는 javax.validation.groups.Default 유형이어야 하며, group 클래스 사이에서 계층 관계를 가질 수 있다. 그리고 하위 계층 그룹에 대해 Validation Check 요청이 있을 경우 상위 계층에 대한 Validation Check도 함께 이루어지게 된다. groups 속성값이 정의되지 않은 경우 Default group으로 간주된다.

```

public interface Draft extends Default {
}

```

위와 같이 코드가 구성된 경우 Validator의 validate() 메소드 호출시 group 정보를 인자로 전달하면 해당 group에 속한 속성 정보에 대해서만 Validation Check가 수행된다.

```

Set<ConstraintViolation<Movie>> constraintViolations = validator.validate(movie,
    Draft.class);
System.out.println("the number of constraint violation is " +
    constraintViolations.size());

Iterator<ConstraintViolation<Movie>> iterator = constraintViolations.iterator();

while (iterator.hasNext()) {
    ConstraintViolation<Movie> constraintViolation = iterator.next();
    // ...
}

```

위에서 언급한 groups 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-validation- groups를 통해 다운로드받을 수 있다.

- message : Validation Error가 발생하였을 경우 표현되는 메시지를 정의하는데 사용된다. 기본적으로 사용중인 Validator를 포함하는 라이브러리 내에 포함된 메시지 리소스 파일로부터 해당 Annotation의 {fully-qualified class name}.message에 해당하는 메시지 값을 추출하게 된다. 예를 들어, @NotNull Check시 예러가 발생하면 **javax.validation.constraints.NotNull.message**에 해당하는 메시지가 표현될 것이다. 기본적으로 제공되는 메시지가 아닌 다른 메시지를 표현해주고 싶을 경우에는 message의 속성값으로 신규 메시지 key를 정의하면 된다. 그리고 클래스패스 상위에 해당 key와 이에 대한 메시지를 포함하고 있는 메시지 리소스 파일을 정의한다.

메시지 리소스 파일에 대해서는 기본적으로 국제화가 지원되며, Hibernate Validator의 경우 기본적으로 영어,불어,독일어 형태의 메시지 리소스 번들을 제공하고 있는데 만일 다른 언어로 구성된 메시지 리소스 파일을 추가하고자 원한다면 클래스패스 내에 org/hibernate/validator/ValidationMessages_{locale}.properties 파일을 추가하고 JSR-303 Annotation 각각에 대한 메시지를 정의하도록 한다.

13.2.3. Custom Constraints

JSR-303에서 기본적으로 제공하는 Annotation만으로 특정 도메인 클래스의 속성값에 대한 Validation Check가 수행되기 어려운 경우 프로젝트에 적합한 Custom Constraints를 정의할 수 있다. Custom Constraints를 활용하기 위해서는 Custom Annotation과 Custom Validator 구현이 이루어져야 한다. 다음은 전화번호 속성에 대한 Validation Check를 위해 신규 정의한 Telephone.java 클래스의 일부이다.

```
@Target( { ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = TelephoneValidator.class)
@Size(min = 12, max = 13)
public @interface Telephone {
    String message() default
        "{org.anyframe.sample.validation.constraint.Telephone.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

위 코드에 의하면 @Telephone은 Method, Field에 대해 정의 가능하며 런타임시에 적용된다. 그리고 도메인 클래스 내에 @Telephone이 부여된 속성을 만나면 TelephoneValidator가 초기화되어 Validation Check를 수행할 것이다. 또한 @Size Annotation 정의가 추가되어 있어서 @Telephone은 기본적으로 Size에 대해서도 제약하게 된다. @Telephone은 JSR-303 Spec.에서 정의한 기본 속성(message, groups, payload) 외에 추가 속성을 포함하고 있지는 않다.

message 속성의 경우 기본값을 org.anyframe.sample.validation.constraint.Telephone.message으로 정의하고 있으므로 @Telephone에 대한 Validation Check 관련 Error가 발생한 경우 클래스패스 최상위의 ValidationMessages.properties 파일로부터 org.anyframe.sample.validation.constraint.Telephone.message을 key로 하는 메시지가 출력될 것이다. 다음은 ValidationMessages.properties 파일의 내용이다.

```
org.anyframe.sample.validation.constraint.Telephone.message=must match "0000-000(or
0000)-0000" (max 13)
```

다음은 @Telephone Annotation에 대해 Validation Check를 수행할 TelephoneValidator.java 파일의 일부이다. 다음 코드에서와 같이 Custom Validator는 javax.validation.ConstraintValidator 인터페이스를 implements해야 하며 Validation Check 로직을 수행할 isValid()라는 메소드를 구현해주어야 한다.

```
public class TelephoneValidator implements ConstraintValidator<Telephone, String> {
    private java.util.regex.Pattern pattern = java.util.regex.Pattern
        .compile("[0-9]\\d{2}-(\\d{3}|\\d{4})-\\d{4}$");

    public void initialize(Telephone annotation) {
    }

    public boolean isValid(String value, ConstraintValidatorContext context) {
        if (value == null || value.length() == 0) {
            return true;
        }
        Matcher m = pattern.matcher(value);
        return m.matches();
    }
}
```

TelephoneValidator는 Regular Expression을 이용하여 전화번호에 대한 패턴을 정의해 두고 이 패턴과 동일하지 않을 경우 Validation Error를 발생하게 된다.

다음은 앞서 정의한 @Telephone 정의를 포함하고 있는 도메인 클래스 Movie.java 파일의 일부이다.

```
public class Movie implements Serializable {

    private static final long serialVersionUID = 1L;

    private String movieId;

    @Telephone
    private String telephone;

    // ...

}
```

위에서 언급한 Custom Constraint 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-validation-custom를 통해 다운로드받을 수 있다.

13.2.4. Declarative Validating

Spring MVC 2.5 이전에서는 앞서 언급한 바와 같이 Spring Validator를 구현하고 이를 특정 Controller의 Validator로 직접 지정해 주어야만 Validation Check가 이루어졌었다. 그러나 Spring 3 이후부터는 Controller 메소드의 입력 인자에 대해 @Valid라는 Annotation을 부여함으로써 해당 메소드 호출 전에 자동적으로 Validation Check가 이루어질 수 있도록 지원한다. 다음은 Core Plugin 설치로 추가된 ~/core/moviefinder/web/MovieController.java 클래스 내에 정의된 create() 메소드의 일부이다.

```
@RequestMapping(params = "method=create")
public String create(..., @Valid Movie movie, BindingResult results, ...) throws Exception {

    if (results.hasErrors()) {
        return "coreViewMovie";
    }

    // ...

}
```

위 메소드의 경우, 사용자의 입력값을 Movie 객체로 매핑할 때 Validation Check가 이루어지게 되고 결과값은 BindingResult 객체에 담겨지게 된다. 따라서 입력 인자로 전달된 BindingResult 객체 내에 Validation Error가 존재하는 경우 입력 화면으로 되돌아가도록 로직을 구성하면 된다.

또한 Spring에서 제공하는 <form:errors>를 활용하면 Validation Error를 입력 화면에 표현해 줄 수 있게 된다. 다음은 Core Plugin 설치로 추가된 webapp/WEB-INF/jsp/core/moviefinder/movie/form.jsp 파일의 일부로 title, director 필드에 입력된 값이 유효하지 않을 경우 <form:errors>를 이용하여 표현해 줄 수 있도록 정의하고 있음을 알 수 있다.

```
<tr>
    <td width="150" class="ct_td"><spring:message code="movie.title" />&nbsp;&nbsp;*</td>
    <td bgcolor="D6D6D6" width="1"></td>
    <td class="ct_write01">
        <form:input path="title" cssClass="ct_input_g" cssErrorClass="text medium error"
        size="40" maxlength="50" />
        <form:errors path="title" cssClass="errors" />
    </td>
</tr>
<tr>
    <td height="1" colspan="3" bgcolor="D6D6D6"></td>
</tr>
<tr>
    <td width="150" class="ct_td"><spring:message code="movie.director" />&nbsp;&nbsp;*</td>
    <td bgcolor="D6D6D6" width="1"></td>
    <td class="ct_write01">
```

```

        <form:input path="director" cssClass="ct_input_g" cssErrorClass="text medium error"
        size="40" maxLength="50" />
        <form:errors path="director" cssClass="errors" />
    </td>
</tr>

```

끝으로 선언적인 Validation Check를 위해서는 Validator 지정을 위한 속성 정의가 필요하다. Spring에서는 이를 위해 3가지 방법을 제공한다.

- Spring 3에서 새롭게 선보이는 mvc namespace를 활용하는 것으로 다음과 같이 정의된 경우 클래스패스로부터 Hibernate Validator와 같은 JSR-303 Validator 구현체가 자동으로 검색되어 모든 @Controller에 적용된다.

```
<mvc:annotation-driven />
```

- Spring 3에서 새롭게 선보이는 mvc namespace를 활용하되 특정 Validator를 지정하는 것으로 지정된 Validator가 모든 @Controller에 적용된다.

```
<mvc:annotation-driven validator="..." />
```

- Controller 클래스 내에 @InitBinder 메소드를 정의하고 해당 메소드의 입력 인자로 전달된 Binder에 특정 Validator를 셋팅하는 것으로, 이 경우 셋팅된 Validator가 특정 Controller에만 적용된다.

```

@Controller
public class MovieController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(new CustomValidator());
    }

    // ...
}

```

13.2.5. Programmatic Validating

Spring에서는 Validation Check가 필요한 경우에 Hibernate Validator와 같은 JSR-303 Validator 구현체를 실행시킬 수 있도록 하기 위해 LocalValidatorFactoryBean 클래스를 제공한다. LocalValidatorFactoryBean은 클래스패스 내에 JSR-303 구현체와 관련된 라이브러리를 검색하여 Validator를 자동으로 검색해주는 역할을 수행한다. 따라서 LocalValidatorFactoryBean을 Bean으로 정의하고 특정 클래스에서 이 Bean을 참조하여 Validation Check를 수행하면 된다.

```

<bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" />

```

```

@Service
public class MovieServiceImpl implements MovieService {
    /**Inject a reference to javax.validation.validator if you prefer to work with the
    JSR-303 API directly.
    * Inject a reference to org.springframework.validation.Validator if your bean requires
    the Spring Validation API
    */
    @Inject
    private Validator validator;

    public void create(Movie movie){
        validator.validate(movie);
        // ...
    }
}

```

```
}
}
```

13.3.Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, `mvn compile exec:java -Dexec.mainClass=...`이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, `src/main/java` 폴더 하위의 `Main.java`를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 `Run As > Java Application`을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
anyframe-sample-validation-payload.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-validation-payload.zip]
anyframe-sample-validation-groups.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-validation-groups.zip]
anyframe-sample-validation-custom.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-validation-custom.zip]

14.Data Binding and Type Conversion

Spring에서 타입 변환이 발생하는 영역은 크게 2가지이다. 하나는 Bean 정의 XML에서 `<property />`를 이용해 설정한 값을 실제 Bean 객체의 Property에 바인딩 시킬 때인데, XML에 String으로 정의한 값을 해당 Property의 타입으로 변환해서 셋팅해야한다.

예를 들어, Movie 클래스가 다음과 같이 정의되어 있고,

```
public class Movie {
    String id;
    String name;
    int ticketPrice;
}
```

'movie' Bean을 아래와 같이 정의했다고 하면,

```
<bean id="movie" class="sample.Movie">
    <property name="name" value="Avatar"/>
    <property name="ticketPrice" value="7500"/>
</bean>
```

'name'이라는 Property는 같은 String 타입이기 때문에 문제가 없지만, 'ticketPrice'의 경우 String으로 작성된 '7500'값을 int 타입의 7500으로 변환하여 바인딩 해야한다.

타입 변환이 발생하는 다른 한가지 경우는, 아래 코드 예와 같이 HTTP Request 파라미터로 들어온 사용자 입력 값들을 'Movie'라는 Model 객체에 바인딩시킬 때이다. 여기서도 마찬가지로 문자열로 표현된 값을 특정 타입으로 변환하는 과정이 필요하다.

```
@RequestMapping("/movies/new", method=RequestMethod.POST)
public String create(@ModelAttribute Movie movie, BindingResult results) {

    this.moviesService.create(movie);
    status.setComplete();

    return "redirect:/movies";
}
```

또한 단순히 타입의 변환이 아니라, 사용자가 보는 View에서 값에 "\$45.22"와 같은 특정 Format이 적용되어 변환되어야 하는 경우도 종종 있다.

이 장에서는 이러한 타입 변환을 위해서 Spring에서 지원하고 있는 기술들에 대해서 자세히 알아보도록 하겠다.

14.1.PropertyEditor

Spring에서는 위에서 언급한 타입 변환을 위해서 기본적으로 JavaBeans 표준에서 제공하는 PropertyEditor를 사용해왔다. PropertyEditor는 String과 특정 타입 객체 간의 변환 로직을 구현할 수 있는 인터페이스이다.

14.1.1.Implementing Custom Editor

타입 변환시 호출되는 PropertyEditor의 메소드는 setValue()/getValue(), setAsText()/getAsText() 4가지이다. PropertyEditorSupport를 상속받아서 setAsText()/getAsText() 메소드만 오버라이드하면 특정 타입 변환을 위한 PropertyEditor를 구현할 수 있다.

Spring에서 제공하고 있는 CustomBooleanEditor 코드를 조금 살펴보면, 아래와 같이 setAsText() 메소드에는 String값을 받아서 boolean값으로 변환하여 setValue() 해주는 로직이 구현되어 있고, getAsText() 메소드에는 getValue() 호출해서 가져온 값을 String으로 변환하여 리턴하는 로직이 구현되어 있다.

```

@Override
public void setAsText(String text) throws IllegalArgumentException {
    String input = (text != null ? text.trim() : null);
    if (this.allowEmpty && !StringUtils.hasLength(input)) {
        setValue(null);
    } else if (this.trueString != null && input.equalsIgnoreCase(this.trueString)) {
        setValue(Boolean.TRUE);
    } else if (this.falseString != null && input.equalsIgnoreCase(this.falseString)) {
        setValue(Boolean.FALSE);
    } // 중략
    } else {
        throw new IllegalArgumentException("Invalid boolean value [" + text + "]");
    }
}

@Override
public String getAsText() {
    if (Boolean.TRUE.equals(getValue())) {
        return (this.trueString != null ? this.trueString : VALUE_TRUE);
    } else if (Boolean.FALSE.equals(getValue())) {
        return (this.falseString != null ? this.falseString : VALUE_FALSE);
    } else {
        return "";
    }
}
}

```

14.1.2.Default PropertyEditors

위에서 본 CustomBooleanEditor와 같이 Spring에서는 기본 타입에 대해서 이미 구현해놓은 여러가지 Built-in PropertyEditor들을 제공한다. Built-in PropertyEditor들은 모두 **org.springframework.beans.propertyeditors** 패키지 하위에 존재한다.

ClassEditor, FileEditor, InputStreamEditor, LocaleEditor, PropertiesEditor 등의 Built-in PropertyEditor들의 이름에서 볼 수 있듯이 Built-in PropertyEditor들은 변환할 타입에 'Editor'라는 이름을 붙인 클래스들이다. CustomNumberEditor와 같이 사용자가 Customizing이 가능한 PropertyEditor에는 'Custom'이라는 접두어가 붙기도 한다. 이들은 모두 디폴트로 등록되어 내부적으로 사용되지만, **CustomDateEditor**와 **StringTrimmerEditor**는 디폴트로 등록되지 않기 때문에, 사용이 필요한 경우에는 반드시 직접 코드에서 등록해 주어야 한다.

14.1.3.Register Custom Editor

기본적으로 Spring에서는 Built-in PropertyEditor들을 미리 등록해놓고 사용하고 있다. 이외에 추가로 Custom Editor 등록이 필요한 경우 따로 등록을 해주어야 하는데, 이 장에서는 Custom PropertyEditor를 어떻게 등록할 수 있는 지에 대해서 알아보도록 하겠다. Spring MVC에서 사용자가 추가로 개발한 Custom PropertyEditor를 등록하는 방법에는 아래와 같이 3가지가 있다.

- 개별 컨트롤러에 적용

Controller에서 @InitBinder annotation을 이용하여 PropertyEditor 등록하는 메소드 정의

```

@InitBinder
public void initBinder(WebDataBinder binder) {
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
    binder.registerCustomEditor(Date.class, new CustomDateEditor(df, false));
}

```

- 전체 컨트롤러에 적용

어플리케이션 전반에서 많이 사용되는 Custom PropertyEditor의 경우 **WebBindingInitializer** 이용

1. WebBindingInitializer를 구현한 클래스 생성

```
public class ClinicBindingInitializer implements WebBindingInitializer {
    @Autowired
    private Clinic clinic;

    public void initBinder(WebDataBinder binder, WebRequest request) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat,
            false));
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(false));
        binder.registerCustomEditor(PetType.class, new PetTypeEditor(this.clinic));
    }
}
```

2. AnnotationMethodHandlerAdapter에 webBindingInitializer 속성을 이용해서 설정

```
<bean
    class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
    <property name="webBindingInitializer">
        <bean
            class="org.springframework.samples.petclinic.web.ClinicBindingInitializer" />
        </property>
    </bean>
```

- 여러 개의 PropertyEditor를 여러 컨트롤러에 적용

다수의 컨트롤러에서 자주 사용되는 여러 개의 Custom PropertyEditor 셋트로 관리할 경우 **PropertyEditorRegistrar** 이용

1. PropertyEditorRegistrars를 구현한 클래스 생성

```
package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {
    public void registerCustomEditors(PropertyEditorRegistry registry) {

        // 새로운 PropertyEditor 인스턴스 생성
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // 필요한 Custom PropertyEditor들 추가
    }
}
```

2. 구현한 Custom PropertyEditorRegistrar를 Bean으로 등록

```
<bean id="customPropertyEditorRegistrar"
    class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>
```

3. @InitBinder를 이용하여 Controller에서 사용

```
@Inject
private final PropertyEditorRegistrar customPropertyEditorRegistrar;

@InitBinder
public void initBinder(WebDataBinder binder) {
    this.customPropertyEditorRegistrar.registerCustomEditors(binder);
}
```

}

14.1.4.PropertyEditor의 단점

PropertyEditor는 기본적으로 String과 특정 타입 간의 변환을 지원한다. PropertyEditor는 변환 과정 중에, 변환하려고 하는 Object나 String을 PropertyEditor 객체에 잠깐 저장하였다가 변환하기 때문에, 여러 Thread에서 동시에 사용하는 경우, 변환 도중에 가지고 있던 값이 변경되어 엉뚱한 변환 값을 전달할 수도 있다. 이런 이유에서 PropertyEditor는 Thread-Safe하지 않기 때문에, Singleton Bean으로 사용하지 못하고 위에서 봤던 예제 코드에서처럼 항상 'new'를 통해서 새로 생성해야 한다.

14.2.Spring 3 Type Conversion

앞서 언급했듯이 JavaBeans의 표준인 PropertyEditor에는 몇가지 단점이 존재한다. 또한 Spring 내부적으로도 한쪽이 String으로 제한된 타입 변환이 아니라 좀 더 일반적인 타입 변환이 요구되기 시작했다. 그래서 Spring 3에서는 PropertyEditor의 단점을 극복하고 내부적으로 타입 변환이 일어나는 모든 곳에서 사용할 수 있는 범용적인 Type Conversion System을 내놓았다. 이와 관련된 클래스들은 모두 `org.springframework.core.convert` 패키지 하위에 존재한다. 이 장에서는 Spring 3에서 소개한 Type Conversion 서비스의 사용방법에 대해서 자세히 알아보도록 하겠다.

14.2.1.Implementing Converter

Spring 3에서는 Converter 구현을 위해서 다음과 같이 여러가지 API를 제공하고 있다.

- Converter

Spring 3 Type Conversion 시스템에서 타입 변환을 실제 담당하는 객체는 Converter이다. Converter를 작성하려면 Spring에서 제공하는 **`org.springframework.core.convert.converter.Converter<S, T>`** 인터페이스를 구현하면 된다. Generics를 이용해서 Converter를 정의하므로 Run-time Type-Safety를 보장해준다.

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
    T convert(S source);
}
```

Converter 인터페이스에서 구현해야 할 메소드는 `convert()` 메소드 하나이다. 즉 PropertyEditor와는 달리 단방향 타입 변환만 제공한다. 'S'에는 변환 전인 Source 타입을 명시하고, 'T'에는 변환 할 Target 타입을 명시한다. Converter 객체가 변환과 관련된 상태 값을 저장하지 않기 때문에 Converter를 Singleton Bean으로 등록하여 Multi-thread 환경에서도 안전하게 사용할 수 있다.

다음은 Converter를 구현한 예제 코드이다.

```
final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }
}
```

- ConverterFactory

클래스 계층으로 묶을 수 있는 `java.lang.Number`나 `java.lang.Enum`과 같은 타입 변환 로직을 한 곳에서 관리하고자 하는 경우, 아래의 ConverterFactory 인터페이스의 구현클래스를 작성하면 된다..

```
package org.springframework.core.convert.converter;
```



```
public interface ConverterFactory<S, R> {
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

여기서 'S'에는 변환 전인 Source 타입을 명시하고, 'R'에는 변환할 클래스들의 상위 베이스 클래스를 명시한다. 그리고 getConverter() 메소드를 구현하는데, 이 때, 'T'는 'R'의 하위 클래스 타입이 될 것이다.

다음은 ConverterFactory의 구현클래스 예이다. (Spring에서 제공하는 StringToNumberConverterFactory이다.)

```
final class StringToNumberConverterFactory implements ConverterFactory<String, Number> {

    public <T extends Number> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToNumber<T>(targetType);
    }

    private static final class StringToNumber<T extends Number> implements
        Converter<String, T> {

        private final Class<T> targetType;

        public StringToNumber(Class<T> targetType) {
            this.targetType = targetType;
        }

        public T convert(String source) {
            if (source.length() == 0) {
                return null;
            }
            return NumberUtils.parseNumber(source, this.targetType);
        }
    }
}
```

- GenericConverter

또한, 두 가지 이상의 타입 변환을 수행하는 Converter를 개발하고자 하는 경우에는 **GenericConverter** 인터페이스를 구현하면 된다. 여러개의 Source/Target 타입을 지정할 수 있고, Source나 Target 객체의 Field Context(Field에 적용된 Annotation이나 Generics 등을 포함한 Field와 관련된 모든 정보)를 사용할 수 있기 때문에 유연한 Converter이긴 하지만, 그만큼 구현하기가 어렵고 복잡하다. 일반적으로 **Converter**나 **ConverterFactory**만으로 커버할 수 있는 기본적인 변환에는 사용하지 않는 것이 좋다.

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

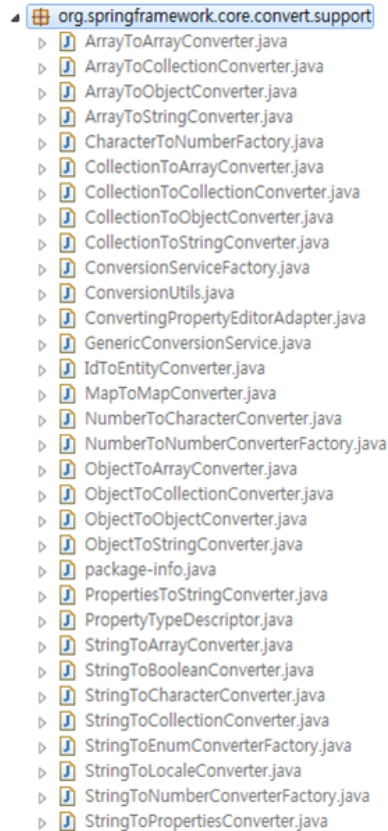
실제 GenericConverter 구현 모습을 보고 싶다면, Spring에서 제공하는 Built-in Converter 중 하나인 org.springframework.core.convert.support.ArrayToCollectionConverter 코드에서 확인할 수 있다.

- ConditionalGenericConverter

만약 어떤 조건을 만족하는 경우에만 변환을 수행하는 Converter를 개발할 경우는 **ConditionalGenericConverter** 인터페이스 구현클래스를 작성한다. 참조할 수 있는 구현 예는 Spring의 org.springframework.core.convert.support.IdToEntityConverter 이다.

14.2.2.Default Converter

Spring에서는 Converter도 PropertyEditor처럼 기본적인 타입들에 대해서 이미 구현해놓은 Built-in Converter들을 제공한다. Built-in Converter들은 모두 **org.springframework.core.convert.support** 패키지 하위에 존재한다.



14.2.3.Register Converter

사용자 필요에 의해서 추가로 개발한 Custom Converter들을 사용하려면 Converter도 역시 PropertyEditor처럼 등록이 필요하다. 한가지 다른 점은 각각의 Converter를 개별적으로 등록하는 것이 아니라, 모든 Converter를 가지고 변환 작업을 처리하는 **ConversionService**를 Bean으로 등록한 후, ConversionService Bean을 필요한 곳에서 Inject 받아서 사용한다는 것이다.

```
package org.springframework.core.convert;

public interface ConversionService {

    boolean canConvert(Class<?> sourceType, Class<?> targetType);

    <T> T convert(Object source, Class<T> targetType);

    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
}
```

실제 Run-time시에 Converter들의 변환 로직은 이 ConversionService에 의해서 실행된다. 기본적으로 Spring에서 사용되는 ConversionService 구현 클래스는 **GenericConversionService**이다. 대부분의 **ConversionService** 구현 클래스는 **Converter** 등록 기능을 가지고 있는 **ConverterRegistry**도 구현하고 있다.

- **ConversionService Bean 정의 시 'converters' 속성 이용**

ConversionService 구현클래스인 GenericConversionService는 ConversionServiceFactoryBean을 이용해서 Bean으로 등록할 수 있다. **ConversionServiceFactoryBean**이 가진 **'converters'** 속성을 이용하면 **Custom Converter**를 추가할 수도 있다.

다음은 ConversionServiceFactoryBean을 사용하여 ConversionService를 Bean으로 정의한 모습이다.

```
<bean id="conversionService"
  class="org.springframework.context.support.ConversionServiceFactoryBean">
  <!-- 추가할 Custom Converter를 설정 -->
  <property name="converters">
    <list>
      <bean class="org.anyframe.sample.moviefinder.StringToFilmRatingConverter" />
      <bean class="org.anyframe.sample.moviefinder.FilmRatingToStringConverter" />
    </list>
  </property>
</bean>
```

ConversionServiceFactoryBean은 ConversionServiceFactory 클래스를 이용해서 디폴트 Converter들을 GenericConversionService에 등록하고, 'converters' 속성을 통해 추가된 Converter들을 등록한다.



'conversionService'이라는 Bean 이름은 Spring에게 양보!

Spring 3에서는 타입 변환을 위해 Run-time 시에 사용되는 ConversionService Bean을 'conversionService'라는 이름으로 찾는다. 따라서 다른 용도의 Bean을 'conversionService'라는 이름으로 등록해서는 안된다.

14.2.4. ConversionService 사용하기

앞서 PropertyEditor는 매번 new 키워드를 이용해서 매번 인스턴스를 새로 생성해야만 했기 때문에 개별 컨트롤러 적용방법과 전체 컨트롤러 적용방법이 달랐었지만, Converter의 경우는 모든 Converter들을 가지고 있는 ConversionService를 Singleton Bean으로 등록해서 사용하기 때문에 아래와 같이 개별 컨트롤러에서 사용하는 것과, WebBindingInitializer 구현클래스를 이용해서 전체 컨트롤러에서 적용하는 것이 차이가 없다.

```
@Inject
private ConversionService conversionService;

@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.setConversionService(this.conversionService);
}
```

따라서 WebBindingInitializer를 구현한 클래스를 이용하여 하나의 설정으로 등록하는 것이 편리하다. Spring에서는 WebBindingInitializer를 직접 구현하지 않고 선언적인 설정만으로도 WebDataBinder의 설정을 초기화할 수 있게 해주는 **ConfigurableWebBindingInitializer** [<http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/bind/support/ConfigurableWebBindingInitializer.html>]를 제공한다.

아래와 같이 설정하기만 하면 Custom Converter들이 추가된 ConversionService가 타입 변환 시에 사용될 것이다.

```
<!-- AnnotationMethodHandlerAdapter에 webBindingInitializer DI -->
<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
  <property name="webBindingInitializer" ref="webBindingInitializer" />
</bean>
```

```

<!-- 사용자가 변경한 conversionService를 WebBindingInitializer 구현체에 DI -->
<bean id="webBindingInitializer"
      class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
    <property name="conversionService" ref="conversionService" />
</bean>

<!-- Custom Converter들을 추가한 conversionService Bean 정의 -->
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class="org.anyframe.sample.moviefinder.StringToFilmRatingConverter" />
            <bean class="org.anyframe.sample.moviefinder.FilmRatingToStringConverter" />
        </list>
    </property>
</bean>

```

위와 같은 복잡한 설정을 쉽고 간편하게 할 수 있도록 **Spring 3**에서는 **mvc** 네임스페이스를 제공한다.

<mvc:annotation-driven>에 대한 자세한 내용은 본 매뉴얼 **Spring MVC >> Configuration**에서 **Configuration Simplification** 내용을 참고하기 바란다.

14.3.Spring 3 Formatting

지금까지 설명한 Conversion System은 Spring에서 범용적인 사용을 목적으로 만들어졌다. Spring 컨테이너에서 Bean의 Property 값을 셋팅할 때, Controller에서 데이터를 바인딩할 때는 물론이고 SpEL에서 데이터 바인딩 시에도 이 Conversion System을 사용한다.

Conversion System은 하나의 타입에서 다른 타입으로의 변환 로직을 구현할 수 있는 일관성있는 API를 제공한다. 그러나 실제로 사용자 UI가 존재하는 어플리케이션에서는 단순한 타입 변환만이 아니라, 날짜나 통화 표현같이 특정 Format을 객체의 값에 적용하여 String으로 변환해야 하는 경우가 종종 있다. 범용적인 용도로 만들어진 Converter에는 이러한 Formatting에 대한 처리 방법이 명시되어있지 않다.

그래서 Spring 3에서는 다음과 같은 **Formatter API**를 제공한다.

```
package org.springframework.format;
```

```
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

```
public interface Printer<T> {
    String print(T fieldValue, Locale locale);
}
```

```
import java.text.ParseException;

public interface Parser<T> {
    T parse(String clientValue, Locale locale) throws ParseException;
}
```

14.3.1.Implementing Formatter

Formatter를 개발하기 위해서는 위의 Formatter 인터페이스를 구현하여야 한다. print() 메소드에서 format을 적용하여 출력하는 로직을 구현하고, parse() 메소드에는 format이 적용된 String 값을 분석해서 객체 인스턴스로 변환하는 로직을 구현하면 된다. 위의 인터페이스 정의에서 볼 수 있듯이, Locale 정보도 함께 넘겨주기 때문에 Localization 적용도 쉽게 처리할 수 있다.

다음은 구현된 Formatter 예제 코드이다.

```
public final class DateFormatter implements Formatter<Date> {

    private String pattern;

    public DateFormatter(String pattern) {
        this.pattern = pattern;
    }

    public String print(Date date, Locale locale) {
        if (date == null) {
            return "";
        }
        return getDateFormat(locale).format(date);
    }

    public Date parse(String formatted, Locale locale) throws ParseException {
        if (formatted.length() == 0) {
            return null;
        }
        return getDateFormat(locale).parse(formatted);
    }

    protected DateFormat getDateFormat(Locale locale) {
        DateFormat dateFormat = new SimpleDateFormat(this.pattern, locale);
        dateFormat.setLenient(false);
        return dateFormat;
    }
}
```

14.3.2.Default Formatter

Spring에서는 편의를 위해서 Formatter 역시 기본적인 Built-in Formatter를 제공하고 있다.

- **DateFormatter**

Spring은 기본적으로 `java.text.DateFormat`을 가지고 `java.util.Date` 객체의 formatting 처리를 하는 `DateFormatter`를 제공한다. (`org.springframework.format.datetime` 패키지) 또한 Spring에서는 강력한 Date/Time 관련 기능을 지원하는 Joda Time Library [<http://joda-time.sourceforge.net/>]를 이용한 formatting도 제공한다. (`org.springframework.format.datetime.joda` 패키지) 클래스패스상에 Joda Time Library가 존재한다면 디폴트로 동작한다.

- **NumberFormatter**

Spring에서는 `java.text.NumberFormat`을 사용한 `java.lang.Number` 객체의 formatting 처리를 위해서 `NumberFormatter`, `CurrencyFormatter`, `PercentFormatter`를 제공하고 있다. (`org.springframework.format.number` 패키지)

일반적으로는 위의 **Formatter**를 직접 사용하기 보다는 아래에서 살펴볼 **Annotation** 기반 **Formatting** 처리 방법, 특히 **Spring**에서 기본적으로 제공하는 **Formatting** 관련 **Annotation** 들을 주로 사용하게 될 것이다.

14.3.3.Annotation 기반 Formatting

다음 섹션에서 살펴보겠지만, 구현된 Formatter는 특정 타입의 변환 시에 사용되도록 등록할 수도 있지만, 특정 Annotation이 적용된 필드의 타입 변환 시에 사용되도록 등록할 수도 있다.

- **Implementation**

Formatting 관련 Annotation을 정의하고 그 Annotation이 적용된 필드의 타입 변환에는 연결되어 있는 특정 Formatter가 사용되도록 하려면 필드에 사용할 **Annotation**과 **AnnotationFormatterFacotry** 구현체를 만들어야 한다.

```
package org.springframework.format;

public interface AnnotationFormatterFactory<A extends Annotation> {

    Set<Class<?>> getFieldTypes();

    Printer<?> getPrinter(A annotation, Class<?> fieldType);

    Parser<?> getParser(A annotation, Class<?> fieldType);

}
```

'A'에는 연결할 Annotation을 명시하고, getFieldTypes()은 해당 Annotation을 적용할 수 있는 필드 타입을 리턴하도록 구현하고, getPrinter()/getParser()는 각각 사용될 Printer와 Parser를 리턴하도록 구현한다.

실제로 Spring에서 제공하고 있는 @NumberFormat의 경우 Annotation과 AnnotationFormatterFacotry가 어떻게 구현되어 있는지 살펴보자.

다음은 @NumberFormat Annotation 구현 코드이다.

```
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface NumberFormat {

    Style style() default Style.NUMBER;

    String pattern() default "";

    public enum Style {
        NUMBER,
        CURRENCY,
        PERCENT
    }

}
```

그리고 다음 코드는 @NumberFormat이 적용된 필드에 어떤 Formatter가 사용되어야 하는지 연결한 AnnotationFormatterFacotry 구현체이다.

```
public final class NumberFormatAnnotationFormatterFactory implements
    AnnotationFormatterFactory<NumberFormat> {

    public Set<Class<?>> getFieldTypes() {
        return new HashSet<Class<?>>(asList(new Class<?>[] {
            Short.class, Integer.class, Long.class, Float.class, Double.class,
            BigDecimal.class, BigInteger.class }));
    }

    public Printer<Number> getPrinter(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    public Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

}
```

```

private Formatter<Number> configureFormatterFrom(NumberFormat annotation, Class<?>
fieldType) {
    if (!annotation.pattern().isEmpty()) {
        return new NumberFormatter(annotation.pattern());
    } else {
        Style style = annotation.style();
        if (style == Style.PERCENT) {
            return new PercentFormatter();
        } else if (style == Style.CURRENCY) {
            return new CurrencyFormatter();
        } else {
            return new NumberFormatter();
        }
    }
}
}

```

이렇게 구현한 Formatter가 실제 Run-time 타입 변환 시에 사용되려면 반드시 등록과정을 거쳐야 한다. Formatter 등록에 대해서는 다음 섹션에서 자세히 알아보도록 하자.

- **Default annotations**

Spring에서 제공하는 **Format** 관련 **Annotation**은 아래와 같이 2가지가 있다.

- **@DateTimeFormat** : java.util.Date, java.util.Calendar, java.util.Long, Joda Time 타입(LocalDate, LocalTime, LocalDateTime, DateTime)의 필드 formatting에 사용 가능

```

public class Movie {
    // 중략
    @DateTimeFormat(pattern="yyyy-MM-dd")
    private Date releaseDate;
}

```

위와 같이 필드에 @DateTimeFormat을 적용하기만 하면 @DateTimeFormat에 연결된 Formatter에 의해서 Formatting이 처리된다.

사용 가능한 속성은 다음과 같다.

Name	Description
style	'S'-Short, 'M'-Medium, 'L'-Long, 'F'-Full 4가지 문자를 날짜에 한글자, 시간에 한글자를 사용해서 두 개의 문자로 만들어 지정. 날짜나 시간을 생략하고자 하는 경우 '.'를 사용 (예: 'S-'). 디폴트 값은 'SS'. Locale 정보를 기반으로 적절한 표현 형식을 적용해 줌
iso	ISO 표준을 사용하고자 하는 경우, @DateTimeFormat(iso=ISO.DATE) 와 같이 지정. ISO.DATE, ISO.DATE_TIME, ISO.TIME, ISO.NONE 사용가능, Locale 정보를 기반으로 적절한 표현 형식을 적용해 줌
pattern	Locale과 상관없이 임의의 패턴을 사용하고자 하는 경우, 'yyyy/mm/dd h:mm:ss a'등의 패턴을 지정

- **@NumberFormat** : java.lang.Number 타입의 필드 formatting에 사용 가능

```

public class Movie {
    // 중략
    @NumberFormat(pattern = "#,##0")
    private int ticketPrice;
}

```

위와 같이 필드에 @NumberFormat을 적용하기만 하면 @NumberFormat에 연결된 Formatter에 의해서 Formatting이 처리된다. java.lang.Number 하위의 클래스인 Byte, Double, Float, Integer, Long, Short, BigInteger, BigDecimal 변환에도 사용할 수 있다.

사용 가능한 속성은 다음과 같다.

Name	Description
style	NUMBER, CURRENCY, PERCENT 중 선택 가능. Locale 정보를 기반으로 적절한 표현 형식을 적용해 줌
pattern	Locale과 상관없이 임의의 패턴을 사용하고자 하는 경우, '#,##0'등의 패턴을 지정

14.3.4.Register Formatter

Converter 영역에서, 등록된 Converter들을 가지고 실제 Run-time시에 타입 변환을 처리하는 역할을 담당하는 것이 GenericConversionService라면, Formatter에서 GenericConversionService와 같은 역할을 담당하는 것은 **FormattingConversionService**이다. **FormattingConversionService**는 **GenericConversionService**를 상속받고 있다.

위에서 살펴본 과정을 통해서 구현한 Formatter를 등록하는 방법은 Converter 등록과는 달리 불편하다. 설정으로 등록할 수 있는 방법은 아직 제공하고 있지 않고, FormattingConversionService를 초기화해주는 FormattingConversionServiceFactoryBean을 상속받은 클래스를 만들어서, installFormatters() 메소드를 오버라이드하여 Custom Formatter를 추가해야한다.

```
public class CustomFormattingConversionServiceFactoryBean extends
    FormattingConversionServiceFactoryBean {

    @Override
    protected void installFormatters(FormatterRegistry registry) {
        super.installFormatters(registry);

        // 필드 타입과 Formatter를 연결하여 등록하는 경우
        registry.addFormatterForFieldType(FilmRatings.class, new FilmRatingsFormatter());

        // Annotation과 Formatter를 연결하여 등록하는 경우
        registry.addFormatterForFieldAnnotation(new
            FilmRatingsFormatAnnotationFormatterFactory());
    }
}
```

위 코드에서 **FormatterRegistry**가 Formatter 등록과 관련된 메소드를 제공하는 것을 확인할 수 있다.

이렇게 확장한 FormattingConversionServiceFactoryBean를 아래와 같이 Bean으로 등록하고, Converter에서처럼 ConfigurableWebBindingInitializer를 이용하여 컨트롤러에서 사용할 수 있도록 설정할 수도 있고,

```
<bean id="conversionService"
    class="org.anyframe.sample.format.CustomFormattingConversionServiceFactoryBean" />
```

아래와 같이 mvc 네임스페이스의 **<mvc:annotation-driven>**를 이용하면 간편하게 설정할 수도 있다.

```
<mvc:annotation-driven conversion-service="conversionService" />

<bean id="conversionService"
    class="org.anyframe.sample.format.CustomFormattingConversionServiceFactoryBean" />
```

<mvc:annotation-driven>만 설정해주면 기본적으로 제공하는 Built-in Converter와 Built-in Formatter, 그리고 Formatting관련 Annotation인 @DateTimeFormat, @NumberFormat을 사용할 수 있다.



PropertyEditor와 Spring 3 Converter 간의 실행 순서

타입변환이 필요한 경우 기본적으로 ConversionService가 등록되지 않으면 Spring은 PropertyEditor를 기반으로 타입 변환을 수행한다. ConversionService가 등록된 경우라고 하더라도 Custom PropertyEditor가 등록된 경우는 Custom PropertyEditor가 우선적으로 적용된다. Even when ConversionService has been registered, Custom PropertyEditor takes priority when Custom PropertyEditor is registered.

* 우선 순위

1. Custom PropertyEditor
2. Converter
3. Default PropertyEditor

15.File Upload

Fileupload Plugin 매뉴얼 [<http://dev.anyframejava.org/docs/anyframe/plugin/optional/fileupload/1.0.3/reference/htmlsingle/fileupload.html>]을 참조한다.

16. Internationalization

Spring MVC에서는 Presentation Layer에서 사용자의 Local에 따른 국제화를 위해 여러가지 LocaleResolver를 제공한다. Request가 들어오면 DispatcherServlet은 LocaleResolver를 통해 사용자의 Locale을 알아내게 되며 RequestContext.getLocale() 메소드를 사용해서 Locale을 확인할 수 있다.

16.1. 다국어 지원 기능

Spring MVC는 다국어를 지원하기 위하여 LocaleResolver를 제공하고 있으며 특정 LocaleResolver를 정의하지 않을 경우 디폴트로 AcceptHeaderLocaleResolver를 이용한다. 사용자들이 원하는 언어를 직접 선택할 수 있도록 구현해야 한다면 CookieLocaleResolver 또는 SessionLocaleResolver를 이용할 수 있다. 웹 어플리케이션의 화면에 출력해줄 메시지 리소스를 추출하기 위해 Spring MVC에서는 MessageSource를 제공하며 이러한 MessageSource에서 추출한 메시지를 화면에 출력해 줄 수 있는 tag 라이브러리를 제공한다. 사용 방법은 아래와 같다.

- Step 1 : properties 파일 작성

각각 언어에 따른 properties파일을 생성하고 출력할 메시지를 작성한다. PropertiesEditor 이클립스 플러그인을 사용하면 쉽게 작성할 수 있다. 다음은 message-user_ko.properties , message-user_en.properties 파일의 일부이다.

- 한글용 (message-user_ko.properties)

```
title.user.form = 당신의 정보를 입력하세요.
```

- 영어용, default용 (message-user.properties, message-user_en.properties)

```
title.user.form = Input your information
```

- Step 2 : MessageSource 정의

다음은 messageSource가 정의되어 있는 context-user.xml 파일의 일부이다.

```
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>classpath:message/message-generation</value>
      <value>classpath:message/message-moviefinder</value>
      <value>classpath:message/message-converter</value>
    </list>
  </property>
  <property name="defaultEncoding">
    <value>UTF-8</value>
  </property>
</bean>
```

- Step 3: JSP에서 message 사용하기

등록한 message를 JSP에서 사용하기 위해 Spring에서 제공하는 태그라이브러리를 선언한 userForm.jsp 파일의 일부이다.

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
```

다음과 같이 <spring:message> 태그를 사용하여 메시지를 출력할 수 있다.

```
<spring:message code="title.user.form"></spring:message>
```

이러한 spring:message 태그의 속성은 다음과 같다.

속성	설명
arguments	부가적인 인자를 넘겨줌. 콤마로 구분된 문자열, 객체 배열, 객체 하나를 넘김.
argumentSeparator	넘겨줄 인자들의 구분자 설정. 기본값은 콤마.
code	lookup할 메시지의 키 지정. 지정하지 않으면 text에 입력한 값 출력.
htmlEscape	html 기본 escape 속성 오버라이딩. 기본값 false.
javaScriptEscape	기본값 false
message	MessageSourceResolvable 인자로 Spring MVC validation을 거친 errors의 메시지를 쉽게 보여줄 때 사용
scope	결과 값을 변수에 지정할 때 변수의 scope 지정 (page, request, session, application)
text	해당 code로 가져온 값이 없을 때 기본으로 보여줄 문자열. 빈 값이면 null 출력.
var	결과 값을 이 속성에 해당하는 문자열에 바인딩 할 때 사용. 빈 값이면 그냥 JSP에 뿌려줌.

16.1.1.LocaleResolver를 이용한 Locale 변경

LocaleResolver를 사용하여 locale을 바꾸고 싶을 때는 정의한 LocaleResolver를 injection 한 후 setLocale()메소드를 통해 locale을 변경해 줄 수 있다. 또한 resolveLocale(request)메소드를 사용하여 현재 request에 셋팅되어 있는 Locale을 알아낼 수 있다.

```
@Controller
public class UserController {
    @Inject
    LocaleResolver localeResolver;

    protected ModelAndView changeLocale(HttpServletRequest request
        , HttpServletResponse response) throws Exception {
        //request parameter "locale"에 사용자가 설정한 locale을 가지고 온다.(ex> en, ko)
        Locale locale = new Locale(request.getParameter("locale"));
        //localeResolver에 locale 셋팅
        localeResolver.setLocale(request, response, locale);
        //셋팅된 locale 확인
        System.out.println("current locale from locale resolver ===== " +
            localeResolver.resolveLocale(request));
        return new ModelAndView("/jsp/result.jsp");
    }
    ... 생략 ...
}
```

16.1.2.LocaleChangeInterceptor를 이용한 Locale 변경

HandlerMapping에 interceptor를 등록하여 특정 locale의 요청을 가로채서 특정 파라미터에 넣어 온 값으로 locale을 지정할 수 있다. 속성 정의 파일 내의 LocaleChangeInterceptor 정의 예는 다음과 같다.

```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>
```

```

<bean id="localeChangeInterceptor"
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
  <property name="paramName" value="locale"/>
</bean>

<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="interceptors">
    <list>
      <ref bean="localeChangeInterceptor"/>
    </list>
  </property>
  <property name="mappings">
    <value>/list.do=getUserListController</value>
  </property>
</bean>

```

위와 같이 설정한 경우, /list.do?locale=en 이라는 요청이 들어오면 어플리케이션의 Locale이 'English'로 변경된다.

16.2.LocaleResolver

위의 다국어 지원 예에서 처럼 Spring MVC에서는 LocaleResolver를 사용하여 Locale을 얻어올 수 있으며 이러한 LocaleResolver 구현체에는 아래와 같은 것들이 있다.

16.2.1.AcceptHeaderLocaleResolver

사용자의 브라우저에서 보내진 request의 헤더에 accept-language부분에서 Locale을 읽어들인다. 사용자의 OS locale을 나타낸다.

```

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver" />

```

AcceptHeaderLocaleResolver는 setLocale() method를 이용한 locale 변경이 불가능하다.

16.2.2.CookieLocaleResolver

사용자의 쿠키에 설정된 Locale을 읽어 들인다. 다음과 같은 속성을 설정할 수 있다.

속성	기본값	설명
cookieName	classname + LOCALE	쿠키 이름
cookieMaxAge	Integer.MAX_INT	쿠키 살려둘 시간. -1로 해두면 브라우저를 닫을 때 없어짐
cookiePath	/	Path를 지정해 주면 해당 하는 path와 그 하위 path에서만 참조

```

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver" >
  <property name="cookieName" value="clientLanguage"/>
  <property name="cookieMaxAge" value="100000"/>
  <property name="cookiePath" value="web/cookie"/>
</bean>

```

16.2.3.SessionLocaleResolver

request가 가지고 있는 session으로 부터 locale 정보를 가져온다.

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />
```



[참고] Encoding 설정

Java 개발 시 영어 이외의 문자를 처리하는 데 있어서 Encoding(인코딩) 설정 문제 때문에 문자가 깨지는 현상을 볼 수 있다. (한글 처리와 관련한 문제로 종종 발생한다.) 컴퓨터에서 영어, 한글 등 문자를 표현하는 다양한 방식이 있는데 이 방식을 처리하는 데 있어서 Encoding/Decoding 설정이 필요하다.

이 매뉴얼에서는 ASCII와 호환이 가능하면서 유니코드를 표현할 수 있는 UTF-8 인코딩 설정을 예로 들어서 설명하도록 한다.

[1] Console 화면에서의 문자 출력을 위한 Encoding 설정

Java 소스 코드 내(*.java)에서 System.out.println("한글 메시지")와 같이 한글을 Console 화면에 출력하려고 할 때, 정상적으로 한글이 출력되지 않는 경우가 있다. 이유는 Java 소스 코드 파일의 인코딩과 컴파일된 파일이 실행되는 실행 환경의 인코딩이 다르기 때문이다. Java 실행 시 디폴트로 OS 환경에 설정되어 있는 인코딩을 기준으로 동작하므로 실행되는 환경의 인코딩 정보를 변경하기 위해서는 -Dfile.encoding="{파일 인코딩 정보}"과 같은 형태의 옵션을 이용할 수 있다.

파일 인코딩을 UTF-8로 설정한 경우, Windows OS 환경에서는 아래와 같이 작성한다. (ex. 웹 어플리케이션을 Tomcat을 이용하여 구동한다면 catalina.bat 파일에 작성한다.) 대부분 한글 Windows OS를 로컬 환경에 설치하므로 아래와 같은 추가 작업을 할 필요가 없는 경우가 많다.

```
set JAVA_OPTS=%JAVA_OPTS% -Dfile.encoding=UTF-8
```

Unix/Linux/Mac OS 환경에서는 아래와 같이 작성한다. (ex. 웹 어플리케이션을 Tomcat을 이용하여 구동한다면 catalina.sh 파일에 작성한다.)

```
JAVA_OPTS="$JAVA_OPTS -Dfile.encoding=UTF-8"
```

Eclipse에서 실행 시에는 아래와 같이 Eclipse Preferences에 설정한 후 실행시키도록 한다.

```
Eclipse Preferences > General > Workspace > Text file encoding 설정을 UTF-8로 설정
```

[2] 웹 화면 개발 시 Encoding 설정

웹 어플리케이션 개발 시 영어 이외의 문자 처리를 위한 Encoding/Decoding 설정 방법은 매우 다양하므로 아래에 제시한 방법은 참고용으로만 살펴보도록 한다. WAS별로 제시되는 Encoding 방식이 다를 수 있고, 실제 코드 개발 시 사용하는 자바 API 혹은 자바스 크립트 메소드 또한 다양하므로 해당 프로젝트 요구 사항에 따라 적용하도록 한다.

[2.1] GET 방식 문자열 처리 시, Servlet Container 설정 혹은 JavaScript 함수 이용

웹 어플리케이션을 Tomcat을 이용하여 구동한다면, 아래와 같이 server.xml 파일 내 Connector 태그에서 URLEncoder 옵션을 UTF-8로 설정해주도록 한다. (URLEncoder 설정 방식은 WAS 별로 다를 수 있으므로 참고하도록 한다.)

```
<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1"
            redirectPort="8443" URLEncoder="UTF-8"/>
```

JavaScript에서 인코딩 기능을 제공해주는 함수로 `encodeURIComponent()`와 `encodeURIComponent()`를 이용할 수 있다.

- `encodeURIComponent()` 함수의 경우 인터넷 주소(URL) 표시에 사용되는 특수 문자들을 제외한 나머지 주소 내용을 인코딩할 때 사용한다. 즉, 인터넷 주소(URL) 전체를 인코딩할 때 사용한다.

- `encodeURIComponent()` 함수의 경우 인터넷 주소(URL) 표시에 사용되는 모든 문자를 인코딩하므로 인터넷 주소(URL) 전체를 인코딩할 때 사용하지 않고, 요청 파라미터 값 하나 하나를 따로 인코딩할 때 사용한다.

GET 방식으로 한글과 같은 문자열을 넘기게 되는 경우 특히 IE(Internet Explorer) 브라우저를 사용할 때 인코딩 설정으로 인한 문제가 발생하는 경우가 많으므로 JSP 페이지 개발 시 GET 방식으로 문자열을 넘겨주는 경우 `encodeURIComponent()` 혹은 `encodeURIComponent()` 함수를 사용하여 인코딩을 해주도록 한다.

[2.2] POST 방식 문자열 처리 시 `CharacterEncodingFilter` 이용

JSP 페이지에서 form 내에 요청 파라미터 값으로 문자열을 설정하여 값을 보내게 되면 POST 방식으로 전달되고 이는 HTTP Request Body에 포함되어 전송된다. 이러한 경우 모든 요청에 대해서 Filter 클래스를 수행하도록 `web.xml` 파일 내에 설정하면 `request.setCharacterEncoding(encoding 정보)` 메소드를 실행시키게 된다. `request.setCharacterEncoding()` 메소드의 경우 Request Body에만 적용하게 되어 있기 때문에 POST 방식에만 적용되는 것이다. `request.setCharacterEncoding()` 메소드를 이용하여 지정한 인코딩 방식이 GET 방식의 QueryString에 의한 파라미터에는 적용되지 않으므로 GET 방식의 파라미터에 영어 이외의 한글 등의 문자열이 있을 경우 위의 [2.1]에서와 같은 별도의 처리가 필요함에 유의하도록 한다.

`web.xml` 파일 내에 `encodingFilter`를 설정해주도록 한다.

```
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</
filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>utf-8</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

17.Exception Handling

Spring MVC에서는 선언적인 Exception Handling을 위해서 ExceptionResolver를 제공한다. 이는 Exception의 종류에 따라 Exception 페이지를 지정해 줄 수 있다. 또한 Controller단에서 try~catch문을 이용하여 발생한 exception에 대한 메시지를 입력 뷰에 다시 출력해 줄 수도 있다.

17.1.특정 error 페이지로 이동하여 에러 메시지 출력

Spring MVC에서는 Exception Handling을 위한 HandlerExceptionResolvers를 제공한다. 이는 특정 exception이 발생했을 때 특정 페이지로 이동시킬 수 있다. 일단, 사용자 exception을 정의해준다.

```
public class UserException extends Exception {
    public UserException(){
        super();
    }
    public UserException(String message){
        super(message);
    }
}
```

이렇게 정의된 사용자 exception을 Controller 단, 또는 Service 단에서 exception을 throw할 수 있다. 다음은 UserException을 throw한 예이다.

```
//입력된 userName이 "test" 가 아닐 경우 UserException을 throw해준다.
if(!a.equals("test"))
    throw new UserException(new String(messageSource.getMessage("error.exception.user",
        new String[]{}, Locale.getDefault())));
```

exception을 throw할 때 messageSource를 사용하여 properties파일에 정의된 "error.exception.user"키 값에 대한 메시지를 출력한다. exception 발생 후 포워딩 될 페이지 정보를 매핑하기 위해서 다음과 같이 HandlerExceptionResolvers를 정의해 준다.

```
<bean id="exceptionResolver"
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="sample.services.UserException">userError</prop>
        </props>
    </property>
    <property name="exceptionAttribute" value="sampleException"/>
    <property name="defaultErrorView" value="error"/>
</bean>
```

위 같이 정의할 경우 Controller 내에서 sample.services.UserException이 발생할 경우 viewResolver에 의해 userError라는 view를 찾게되고 그 view에 에러 메시지를 출력하게 된다. 발생한 Exception은 "sampleException" 이름으로 userError.jsp 페이지에 전달하도록 설정하였다. 만약 exceptionAttribute 속성을 사용하지 않았을 때의 디폴트 값은 "exception"이다. 마지막으로 설정한 defaultErrorView 속성은 앞에서 매핑한 Exception외의 다른 에러가 발생할 경우 error.jsp에 에러메시지를 출력하도록 설정하였다. 간단한 Expression Language를 이용하여 발생한 에러 메시지를 출력할 수 있다.

```
<h3>${sampleException.message}</h3>
```


17.2.에러 페이지에 에러 메시지 출력

UI 계층의 에러 처리 부분에서 위와같은 방법을 사용하게 되면 사용자가 입력했던 값이 모두 사라지게 되는 불편함이 생기게 된다. 이러한 불편을 해소하기 위해서 입력 폼 페이지에서 에러 메시지를 출력하고 사용자가 입력한 값을 유지해 줘야하는데 이는 컨트롤러에서 exception을 직접 처리 해줘야한다. 처리 예는 다음과 같다.

```
protected ModelAndView onSubmit(HttpServletRequest request,
                                HttpServletResponse response, Object command,
                                BindException exception) throws Exception {
    ...중략...
    HelloVO vo = (HelloVO) command;
    ...중략...
    try{
        helloworldService.getMessage(vo);
    }catch (UserException e){
        ModelAndView mav = new ModelAndView(getFormView());
        mav.addObject("user",vo);
        mav.addObject("userException",e);
        request.setAttribute("userException",e);
        return mav;
    }
    return new ModelAndView(getSuccessView(),"vo",vo);
}
protected Object formBackingObject(HttpServletRequest request)
    throws Exception {
    request.setAttribute("user",new sample.services.HelloVO());
    return new HelloVO();
}
```

위의 코드를 보면 getMessage()메소드를 호출할때 UserException을 try~catch로 직접 처리 한 다음 이 exception과 사용자가 입력한 데이터를 ModelAndView를 이용하여 전달하고 있다. 이렇게 전달한 error 메시지는 JSP 파일에서 jstl 태그를 사용하여 출력할 수 있다.

```
<c:if test="${not empty userException}">
<h3><font color="red">Error :
<c:out value="${userException.message}"/></font></h3>
</c:if>
```

17.3.Presentation Layer에서 message key를 이용한 locale 변경

Business Layer에서 BaseException이 발생하였을 때 messageKey를 파라미터로 넘겨주면 Presentation Layer에서 그 messageKey를 받아와 원하는 Locale에 맞게 메시지를 조작할 수 있다.

17.3.1.Business Layer의 BaseException 발생

```
public class UserService implements ApplicationContextAware{

    private static Log logger = LogFactory.getLogger(UserService.class);
    private MessageSource messageSource;

    public UserVO getUser(UserVO userVO) throws Exception{
        logger.debug("\n===== UserService is called =====\n");
```

```

        throw new BaseException(messageSource,"error.test.message"
                                , new Object[] {}, "default message");
        //return userVO;
    }
    public void setApplicationContext(ApplicationContext applicaionContext)
        throws BeansException {
        this.messageSource = applicaionContext;
    }
}

```

위에서 "error.test.message"라는 key 값을 파라미터로 넘겨 주었다.

17.3.2.Presentation Layer에서 꺼낸 message key 값에 새로운 Locale로 셋팅

```

try {
    // call business service

    userVO = userService.getUser(userVO);
    // setting view name
    ModelAndView mav = new ModelAndView("/jsp/user/getUser.jsp");
    mav.addObject(userVO);
    // return a ModelAndView object.
    return mav;
} catch (BaseException e) {
    //발생한 BaseException에서 getMessageKey() 메소드를 통해 message key를 추출한다.
    String messageKey = e.getMessageKey();
    System.out.println("\n messageKey =====" + messageKey
                        + "=====\n");
    //추출한 messageKey를 가지고 ENGLISH 로 케일토 다시 셋팅해 주었다.
    throw new BaseException(messageSource.getMessage(messageKey,
        new String[] {}, Locale.ENGLISH));
}

```

18.Spring Integration

Anyframe 은 Spring MVC를 기반으로 구성되어 있으므로 Spring 프레임워크의 다른 모듈과의 연계가 용이하다. 일반 웹 애플리케이션을 개발할 때 Business Layer의 Business Logic을 이용하여 요청을 처리하게 되는데 이 때 Business Layer를 연계하기 위한 방법은 다음과 같다.

18.1.Listener 등록과 Spring 설정 파일 목록 위치 정의

Spring MVC에서는 DispatcherServlet을 사용하여 WebApplicationContext를 로드하게 된다. 이때 Presentation Layer에서 사용할 Business Layer의 서비스 bean들을 ContextLoaderListener 등록 후 contextConfigLocation으로 Spring 설정 파일 위치를 지정해줌으로써 Presentation Layer에서 Business 서비스 bean들을 호출하여 사용할 수 있다. 다음은 설정 예인web.xml 파일의 일부이다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:/spring/context-*.xml
  </param-value>
</context-param>
<!-- 리스너 등록 -->
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

18.2.Dependency Injection을 통한 Business Service 호출

위와 같이 Listener를 등록하고 Spring 설정 파일 위치를 지정해 주었으면 일반 서비스 호출과 같이 Dependency Injection을 사용하여 Business Service를 호출할 수 있다.

다음은 Annotation을 이용하여 Business Service를 호출한 MovieController.java 파일의 일부이다.

```
@Controller("coreMovieController")
@RequestMapping("/coreMovie.do")
public class MovieController{

    @Inject
    @Named("coreMovieService")
    private MovieService movieService;

    @RequestMapping(params = "method=get")
    public String get(@RequestParam("movieId") String movieId, Model model)
        throws Exception {
        Movie movie = this.movieService.get(movieId);
        if (movie == null) {
            throw new Exception("Resource not found " + movieId);
        }
        model.addAttribute(movie);
    }
}
```

```
    return "core/moviefinder/movie/form";  
  }  
}
```

Spring IoC 컨테이너 Dependency Injection에 대한 자세한 사항은 본 매뉴얼 >> Spring >> IoC(Inversion of Control) >> Dependencies를 참고한다.

18.3.Resources

- 참고자료
 - Spring 4.0.0.RELEASE reference manual - Web MVC framework [<http://static.springsource.org/spring/docs/4.0.0.RELEASE/spring-framework-reference/html/mvc.html>]

IV.Spring MVC Extensions

Anyframe 에서는 개발자들이 보다 프리젠테이션 레이어 개발을 쉽게 할 수 있도록 anyframe namespace와 @Enable* 어노테이션 그리고 Custom tag library를 제공한다. Custom tag library에는 페이지 네비게이션을 JSP단의 java 코드 없이 태그로 개발할 수 있는 Page Navigator 태그가 있다.

19.Configuration Simplification

19.1.<mvc:annotation-driven>

Spring 3.1 에서는 Annotation 기반의 Controller 처리를 위해 반드시 필요한 RequestMappingHandlerAdapter 등록 등의 Spring MVC 관련 설정을 간편하게 할 수 있도록 mvc [http://static.springsource.org/schema/mvc/spring-mvc.xsd] 네임스페이스를 제공하고 있다. mvc 네임스페이스를 통해서 다양한 태그가 지원되고 있는데 그 중, <mvc:annotation-driven>는 HTTP 요청을 Annotation 기반으로 구현된 Controller로 전달해주는 RequestMappingHandlerMapping, RequestMappingHandlerAdapter등을 몇가지 디폴트 설정과 함께 등록해주는 역할을 수행하고 있다.

그런데 <mvc:annotation-driven>를 사용하여 Spring MVC 관련 설정을 하게 되는 경우, RequestMappingHandlerAdapter등의 하위 속성인 synchronizedOnSession 속성값이 'false'로 설정되어 Double Submit 방지 기능을 수행할 수 없게 된다.

따라서, Anyframe에서는 <mvc:annotation-driven>와 동일한 기능을 수행하면서도 Double Submit 방지 기능을 수행할 수 있도록 하기 위해 RequestMappingHandlerAdapter의 하위 속성인 synchronizedOnSession 값을 정의할 수 있는 <anyframe:annotation-driven>를 추가로 제공하고 있다. 다음은 Core Plugin 설치로 생성된 샘플 프로젝트 하위의 src/main/resources/core-servlet.xml 파일의 일부로, <anyframe:annotation-driven />을 활용하여 속성을 정의하고 있음을 알 수 있다.

```
<?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"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:anyframe="http://www.anyframejava.org/schema/mvc"
       xsi:schemaLocation=
           "http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
            http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.1.xsd
            http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
            http://www.anyframejava.org/schema/mvc http://www.anyframejava.org/schema/mvc/anyframe-spring-mvc-5.3.xsd">

    <anyframe:annotation-driven synchronizeOnSession="true"/>

    <!-- 중략 -->
</beans>
```

위 XML에서와 같이 <anyframe:annotation-driven>를 정의하기 위해서는 <beans> 내에 anyframe 네임스페이스에 대한 정의가 반드시 추가되어야 한다.

19.2.@EnableWebMvcAnyframe

Spring은 3.0 부터 Java Based Configuration을 지원해왔으며, 3.1부터는 이를 더욱 발전시켜 @Enable* 형태의 어노테이션 기반 설정 간략화 기능을 제공하고 있다.

Spring은 <mvc:annotation-driven/>에 대응되는 @EnableWebMvc 어노테이션을 제공하고 있다.(스프링에서 제공하는 @Enable* 어노테이션들에 관한 상세한 내용은 여기를 참고하기 바란다.)

이에 Anyframe에서는 Core Plugin 1.0.4 버전부터 <anyframe:annotation-driven/>에 대응되는 @EnableWebMvcAnyframe 어노테이션을 제공하고 있다. @EnableWebMvcAnyframe의 사용예제는 다음과 같다.

```

package org.anyframe.sample.javaconfig.config.web;

import java.util.Locale;

import org.anyframe.spring.config.EnableWebMvcAnyframe;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Controller;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.handler.SimpleMappingExceptionResolver;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvcAnyframe(synchronizeOnSession="true")
@ComponentScan(basePackages = "org.anyframe.sample.javaconfig.moviefinder",
    useDefaultFilters = false, includeFilters = { @ComponentScan.Filter(Controller.class) })
public class WebAppConfig {

    private static final String VIEW_RESOLVER_PREFIX = "/WEB-INF/jsp/";
    private static final String VIEW_RESOLVER_SUFFIX = ".jsp";
    private static final String DEFAULT_ERROR_VIEW = "forward:/sample/common/error.jsp";
    private static final String WARN_LOG_CATEGORY = "controller.logs";

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix(VIEW_RESOLVER_PREFIX);
        viewResolver.setSuffix(VIEW_RESOLVER_SUFFIX);

        return viewResolver;
    }

    @Bean
    public HandlerExceptionResolver exceptionResolver() {
        SimpleMappingExceptionResolver exceptionResolver = new SimpleMappingExceptionResolver();
        exceptionResolver.setDefaultErrorView(DEFAULT_ERROR_VIEW);
        exceptionResolver.setWarnLogCategory(WARN_LOG_CATEGORY);

        return exceptionResolver;
    }

    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver localeResolver = new SessionLocaleResolver();
        localeResolver.setDefaultLocale(new Locale("en_US"));
        return localeResolver;
    }
}

```

본 예제 코드는 Spring MVC 관련 설정을 모아놓은 Web Application Configuration class 이며, @EnableWebMvcAnyframe 어노테이션이 @Configuration과 같이 사용되었고, 세부적으로 synchronizeOnSession 부분의 값을 true로 설정한 것에 주목하기 바란다.

19.3.Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, `mvn compile exec:java -Dexec.mainClass=...`이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, `src/main/java` 폴더 하위의 `Main.java`를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 `Run As > Java Application`을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
anyframe-sample-javaconfig-mvc.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-javaconfig-mvc.zip]

20.Tag library

Anyframe 에서는 개발자들이 자바 코드를 사용하지 않고 보다 쉽게 JSP 페이지를 구현할 수 있도록 다음과 같은 Anyframe Tag Library를 제공한다.

20.1.Page Navigator Tag

Anyframe 에서는 Page 처리에 대한 구현이 편리하도록 page 관련 Tag Library인 Page Navigator Tag를 제공한다. 이 태그를 사용하면 리스트 화면을 출력할 때 Tag Library를 사용하여 간단히 Page Navigator를 출력해줄 수 있다. 이 태그를 사용하기 위해 JSP의 상단에 다음과 같이 anyframe-pagenavigator.tld 파일을 taglib으로 지정해 준다.

```
<%@ taglib uri='/WEB-INF/anyframe-page.tld' prefix='anyframe' %>
```

prefix를 'anyframe'으로 정의할 경우 아래와 같이 태그를 사용할 수 있다.

```
<anyframe:pagenavigator linkUrl="javascript:fncGetUserList(2);"  
  pages="<%=resultPage%>" formName="listForm"/>
```

단, Anyframe에서 제공하는 Page Navigator Tag를 통해 생성되는 Page Navigator의 스타일은 CSS(Cascading Style Sheet) 기반으로 정의된다는 것을 기억해야 한다. 이는 개발자가 Page Navigator의 스타일을 자유롭게 변경할 수 있도록 지원하기 위함이다. (Core 1.0.1 이후)

anyframe을 prefix로 하는 태그로 tag name은 pagenavigator이다 . 이 때 pages라는 attribute는 반드시 org.anyframe.pagination.Page 타입의 객체를 설정해줘야 함에 유의하도록 한다.

V.Id Generation

Idgen Plugin 매뉴얼 [<http://dev.anyframejava.org/docs/anyframe/plugin/optional/idgen/1.6.0/reference/htmlsingle/idgen.html>]을 참조한다.

VI. Logging

개발자가 Log를 출력하기 위해 일반적으로 사용하는 방식은 `System.out.println()`이다. 그러나 이 방식은 간편한 반면에 다음과 같은 이유로 권장하지 않는다.

- `System.out.println`에 대한 호출은 disk I/O동안 동기화(synchronized)처리가 되므로 시스템의 throughput을 떨어뜨린다.
- 기본적으로 stack trace 결과는 콘솔에 남는다. 하지만 시스템 운영중 콘솔을 통해 Exception을 추적하는 것은 바람직하지 못하다.
- 운영시스템에서 시스템 관리자가 `System.out`과 `system.err`에 대하여 '[>null]'(NT의 경우) 혹은 `dev/null`(Unix의 경우)와 같이 매핑을 할 경우 Exception 로그에 대한 출력이 나타나지 않을 수도 있다. 또한 NT 서비스로 실행될 경우 콘솔 자체가 안보일 수도 있다.
- 콘솔 로그를 출력 파일로 리다이렉트 할 지라도, JavaEE App Server가 재 시작할 때 파일이 overwrite될 수도 있다.
- 개발/테스팅 시점에만 `System.out.println`을 사용하고 운영으로 이관하기 전에 삭제하는 것은 좋은 방법이 아니다. 운영시의 코드가 테스트시의 코드와 다르게 동작할 수 있기 때문이다.

따라서, 테스트 코드와 운영 코드를 동일하게 가져가면서 로깅을 선언적으로 관리할 수 있고, 운영시 성능 오버헤드를 최소화할 수 있는 메커니즘이 필요하다. 이런 기능을 위해 Anyframe Framework은 Log4j [<http://logging.apache.org/log4j/>]를 이용하여 로그를 남길 수 있는 방법을 가이드하고자 한다.

21.Configuration

이번 절에서는 log4j.xml 파일을 구성하는 Tag 중, 주로 사용될 일부 Tag에 대해 설명하고자 한다. 보다 자세한 내용에 대해서는 Log4j [<http://logging.apache.org/log4j/>]를 참조하도록 한다. log4j.xml 파일의 root tag인 <log4j:configuration>은 하위에 appender, logger, root등의 tag를 가질 수 있다.

Tag 명	설명	필수 여부
appender	로그가 출력될 대상과 방법을 정의한다. 여러 appender 정의 가능.	N
logger	어플리케이션에서 사용될 Logger를 정의한다. 여러 logger 정의 가능	N
root	모든 logger의 상위 logger를 정의한다.	N

위 표에서 열거한 각 Tag에 대해 보다 자세히 알아보도록 하자.

21.1.appender

Log4j는 다양한 로그 방식을 지원한다. 가장 단순한 Console부터 시작해서 파일, DB, SMTP 등의 방식들을 지원한다.

- **org.apache.log4j.ConsoleAppender** : Console 화면으로 출력하기 위한 Appender. 다음은 log4j.xml 파일 내의 ConsoleAppender에 대한 속성 정의 내용이다.

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %p [%c] - %m%n" />
  </layout>
</appender>
```

- **org.apache.log4j.FileAppender** : 특정 파일에 로그를 출력하기 위한 Appender로 하위에 File, Append 와 같은 parameter를 정의할 수 있다. 다음은 log4j.xml 파일 내의 FileAppender에 대한 속성 정의 내용이다.

```
<appender name="file" class="org.apache.log4j.FileAppender">
  <!-- 로그 파일명 정의를 위한 parameter -->
  <param name="File" value="./logs/file/sample.log"/>
  <!-- 이전 로그 파일에 로그를 덧붙여 쓸 것인지 정의를 위한 parameter -->
  <param name="Append" value="true"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %p [%c] - %m%n" />
  </layout>
</appender>
```

- **org.apache.log4j.RollingFileAppender** : FileAppender는 지정한 파일에 로그가 계속 남으므로 한 파일의 크기가 지나치게 커질 수 있으며 계획적인 로그 관리가 어렵다. 따라서 파일의 크기 또는 파일 백업 인덱스 등의 지정을 통해 특정 크기 이상 파일의 크기가 커지게 되면 기존 파일을 백업 파일로 바꾸고 처음부터 다시 로깅을 시작한다. 하위에 File, Append, MaxFileSize, MaxBackupIndex와 같은 parameter를 정의할 수 있다. 다음은 log4j.xml 파일 내의 RollingFileAppender에 대한 속성 정의 내용이다.

```
<appender name="rollingFile" class="org.apache.log4j.RollingFileAppender">
  <!-- 로그 파일명 정의를 위한 parameter -->
  <param name="File" value="./logs/rolling/sample.log"/>
  <!-- 이전 로그 파일에 로그를 덧붙여 쓸 것인지를 정의하기 위한 parameter -->
  <param name="Append" value="true"/>
```

```

<!-- 로그 파일의 최대 크기를 정의하기 위한 parameter -->
<param name="MaxFileSize" value="1KB"/>
<!-- 로그 파일 백업 인덱스를 정의하기 위한 parameter -->
<param name="MaxBackupIndex" value="2"/>
<layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d %p [%c] - %m%n" />
</layout>
</appender>

```

- **org.apache.log4j.DailyRollingFileAppender** : 로그 파일을 설정한 스케줄에 맞춰 생성하기 위한 Appender로, 매월 첫번째날, 매주 시작일, 매일 자정 및 정오, 매 시간마다 등으로 설정이 가능하다. DailyRollingFileAppender 파일에서 사용할 수 있는 몇가지 날짜 포맷은 다음과 같다.

- `%.yyyy-MM` : 매달 첫번째 날에 로그파일 변경
- `%.yyyy-ww` : 매주 시작시 로그파일 변경
- `%.yyyy-MM-dd` : 매일 자정에 로그파일 변경
- `%.yyyy-MM-dd-a` : 자정과 정오에 로그파일 변경
- `%.yyyy-MM-dd-HH` : 매시간의 시작마다 로그파일 변경
- `%.yyyy-MM-dd-HH-mm` : 매분마다 로그파일 변경

보다 자세한 사항은 Log4j API [<http://logging.apache.org/log4j/1.2/apidocs/index.html>]를 참조한다. 다음은 log4j.xml 파일 내의 DailyRollingFileAppender에 대한 속성 정의 내용이다.

```

<appender name="dailyRollingFile" class="org.apache.log4j.DailyRollingFileAppender">
    <!-- 로그 파일명 정의를 위한 parameter -->
    <param name="File" value="./logs/daily/sample"/>
    <!-- 로그 파일 생성 스케줄을 위한 parameter -->
    <param name="DatePattern" value="%.yyyy-MM-dd"/>
    <!-- 이전 로그 파일에 로그를 덧붙여 쓸 것인지 정의를 위한 parameter -->
    <param name="Append" value="true"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d %p [%c] - %m%n"/>
    </layout>
</appender>

```

- **org.apache.log4j.jdbc.JDBCAppender** : DB에 로그를 출력하기 위한 Appender로 하위에 Driver, URL, User, Password, Sql과 같은 parameter를 정의할 수 있다. 다음은 log4j.xml 파일 내의 JDBCAppender에 대한 속성 정의 내용이다.

```

<appender name="db" class="org.apache.log4j.jdbc.JDBCAppender">
    <!-- JDBC Driver를 정의하기 위한 parameter -->
    <param name="Driver" value="org.hsqldb.jdbcDriver"/>
    <!-- DB URL을 정의하기 위한 parameter -->
    <param name="URL" value="jdbc:hsqldb:hsqldb://localhost/sampledb"/>
    <!-- DB User를 정의하기 위한 parameter -->
    <param name="User" value="sa"/>
    <!-- DB Password를 정의하기 위한 parameter -->
    <param name="Password" value=""/>
    <!-- 로그를 남길때 수행할 쿼리를 정의하기 위한 parameter -->
    <param name="Sql" value="insert into STMR_LOG (msg)
        values('%d %p [%c] - %m%n')"/>
</appender>

```

Appender Layout

로그를 남길때 단순한 메시지 외에도 현재 로그 대상의 쓰레드명, 로그 시간 등 많은 정보들을 조합할 수 있다. Layout에는 org.apache.log4j.HTMLLayout, org.apache.log4j.PatternLayout,

org.apache.log4j.SimpleLayout, org.apache.log4j.xml.XMLLayout 등이 있다. 이중 가장 많이 사용하는 Layout은 PatternLayout으로서 C 함수의 printf처럼 다양한 로그 메시지 조합을 만들어 낼 수 있다.

- %p : debug, info, warn, error, fatal 등의 Priority 출력
- %m : debug(), info(), warn(), error(), fatal() 등의 함수로 지정한 로그 내용 출력
- %d : 로깅 이벤트가 발생한 시간 기록. 출력 포맷은 %d 후의 {}내에 지정된 형태를 따른다. %d{HH:mm:ss, SSS} 라든가 %d{yyyy MMM dd HH:mm:ss, SSS}와 같은 형태로 사용할 수 있다. Java의 SimpleDateFormat의 형식을 따라 정의할 수 있다.
- %t : 로깅 이벤트가 발생한 스레드의 이름 출력
- %% : % 표시를 출력하기 위해 사용
- %n : 플랫폼 종속적인 개행 문자 출력. \r\n 또는 \n이 될 것이다.



log4j-1.2.x 버전 사용 시 유의점

JDBCAppender를 사용할 때 log4j-1.3.x 버전에서는 Connection pool에서 connection을 얻어오지만 log4j-1.2.x 버전에서는 매번 connection 객체를 생성하게 되므로 리소스 문제가 생길 수 있음을 유의한다.

21.2.logger

로깅 이벤트 발생시 같은 이름으로 선언된 logger를 찾아 해당 logger에게 로그 메시지를 보내고 additivity가 true일 경우, 상위 logger에게도 로그 메시지를 보낸다. 다음은 log4j.xml 파일 내의 logger에 대한 속성 정의 내용이다.

```
<!-- 해당 logger명이 anyframe.services로 시작할 경우
      Console에 DEBUG level로 로그를 남긴다.-->
<logger name="anyframe.services">
  <!-- DEBUG, INFO, WARN, ERROR, FATAL, OFF 중 택일 -->
  <level value="DEBUG"/>
  <!-- 여력 appender-ref 정의 가능 -->
  <appender-ref ref="console"/>
</logger>
```

21.3.root

해당 logger가 존재하지 않거나 상위 logger가 존재하지 않을 경우 모든 로그는 root logger의 정책에 따라 출력된다. 다음은 log4j.xml 파일 내의 root에 대한 속성 정의 내용이다.

```
<root>
  <level value="INFO"/>
  <appender-ref ref="console"/>
</root>
```

22.Logging

로그의 내용에 따라 다양한 레벨(DEBUG, INFO, WARN, ERROR, FATAL)로 선택 가능하다. 각각은 메소드 debug(), info(), warn(), error(), fatal()라는 5가지 메소드를 이용해서 로그를 남길 수 있다. 다만 이때 Logger에 지정된 로그 레벨이 있다면, 지정된 로그 레벨 이하의 로깅 이벤트는 무시된다. 따라서 로그도 남지 않는다. 또한, 로그 메시지는 별도 Resource 파일에 정의된 message key를 이용하여 남기면 메시지 변경 및 다국어 지원이 용이하다. 다음에서는 로그 메시지를 남기기 위한 기본 방법과 ResourceBundle을 이용하는 방법에 대해서 알아보려고 한다.

22.1.기본적인 사용 방법

다음은 기본적인 방법을 사용하여 로그 메시지를 남기는 MovieServiceImpl.java 코드의 일부이다.

```
/**
 * src/main/resources/log4j.xml 파일 설정에 따라 Logger명이
 * org.anyframe.sample.logging 패키지에 해당하는 Logger를 찾고,
 * 해당 Logger를 통해 로그 메시지를 남겨보는 테스트
 */
@Service("movieService")
public class MovieServiceImpl implements MovieService {

    public void create(Movie movie) throws Exception {
        MovieService.LOGGER.debug("DEBUG - call create");
        MovieService.LOGGER.warn("WARNING - call create");
        MovieService.LOGGER.error("ERROR - call create");
        movieDao.create(movie);
    }

    public Movie get() throws Exception {
        MovieService.LOGGER.debug("DEBUG - call get");
        MovieService.LOGGER.warn("WARNING - call get");
        MovieService.LOGGER.error("ERROR - call get");
        return movieDao.get();
    }
}
```

22.2.ResourceBundle을 이용하는 방법

특정 서비스의 구현 클래스에서 ResourceBundle을 이용하여 로그 메시지를 남기기 위해서는 다음과 같은 절차를 따르도록 한다.

1. ResourceBundle을 관리하는 기능을 제공하는 MessageSource Bean을 정의한다.

다음은 MessageSource Bean을 정의하고 있는 context-common.xml 파일의 일부이다.

```
<bean name="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <!-- 중략 -->
            <value>message/message-sample</value>
        </list>
        <!-- 중략 -->
    </property>
</bean>
```

2. 특정 서비스의 구현 클래스는 `MessageSource Bean`을 인식하기 위하여 `implements ApplicationContextAware`해야 한다.

다음은 `MessageSource Bean`을 이용하여 로그 메시지를 남기는 `MovieServiceImpl.java`의 일부 코드이다.

```
public class MovieServiceImpl implements MovieService{
    @Inject
    private MessageSource messageSource;
    // 중략
}
```

3. `Logging Service`를 이용하여 로그를 남길 때 `MessageSource Bean`을 이용한다.

다음은 `Service` 구현 클래스에서 사용할 `Logger`를 정의한 인터페이스 클래스 `MovieService.java` 코드이다.

```
public interface MovieService {
    // MovieServiceImpl에서 사용할 Logger 정의
    Log LOGGER = LoggerFactory.getLogger(MovieService.class);
    String greet();
}
```

다음은 `ResourceBundle`을 이용하여 로그 메시지를 남기는 구현 클래스 `MovieServiceImpl.java`의 일부 코드이다.

```
public void greet() {
    // ResourceBundle을 이용하여 로그 메시지를 남긴다. (argument가 없는 경우)
    MovieService.LOGGER.debug(messageSource.getMessage(
        "sample.default.msg", new String[] {}, Locale.getDefault()));
    // ResourceBundle을 이용하여 로그 메시지를 남긴다. (argument가 1개인 경우)
    MovieService.LOGGER.debug(messageSource.getMessage(
        "sample.msg", new String[] { "Anyframe" }, Locale.getDefault()));
}
```

* 위의 코드에서 참조하고 있는 `Resource` 파일 `message-sample.properties`의 내용은 다음과 같이 `key=value` 형태로 정의한다.

```
sample.default.msg=Hello Guest
sample.msg=Hello {0}
```

23.Resources

- 다운로드

다음에서 테스트 DB를 포함하고 있는 hsqldb.zip과 sample 코드를 포함하고 있는 anyframe-sample-logging.zip 파일을 다운받은 후, 압축을 해제한다. 그리고 hsqldb 폴더 내의 start.cmd (or start.sh) 파일을 실행시켜 테스트 DB를 시작시켜 놓는다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, mvn compile exec:java -Dexec.mainClass=...이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, src/main/java 폴더의 anyframe/sample/logging 하위의 Main.java를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 Run As > Java Application을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
hsqldb.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/hsqldb.zip]
anyframe-sample-logging.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-logging.zip]

- 참고자료

- Simple Logging Facade for Java (SLF4J) Home [<http://slf4j.org/>]
- Log4j Home [<http://logging.apache.org/log4j>]
- Log4j API [<http://logging.apache.org/log4j/docs/api/index.html>]
- Short Intruduction to Log4j [<http://logging.apache.org/log4j/docs/manual.html>]
- Log4jdbc Home [<http://code.google.com/p/log4jdbc/>]

VII.Test

Spring 기반의 웹어플리케이션을 개발할 때, Spring Framework와 JUnit4에서 제공하는 Annotation을 사용하여 Testcase를 사용하여 코드를 검증할 수 있다.

24.Service Code Test

이번 절에서는 Business logic 검증을 위한 Testcase 개발 방법에 대해 설명 하고자 한다.

24.1.JUnit

JUnit은 테스트 코드를 쉽게 작성하고, 실행할 수 있도록 제공되는 open source framework이다.

JUnit에서 제공하는 Annotation은 다음과 같다.

Name	Description
@Test	@Test annotation이 설정된 메소드를 JUnit이 자동으로 테스트 코드로 인식해서 실행 시켜준다.
@Before	모든 테스트 메소드를 실행 시키기 전에 실행 할 초기화 로직을 지정한다.
@After	모든 테스트 메소드를 실행 완료한 후에 실행할 소멸화 로직을 지정한다.

테스트 코드 내에서 bean injection 등 Spring 연계를 위해 필요한 설정은 다음과 같다.

- @RunWith(SpringJUnit4ClassRunner.class) : @RunWith가 설정된 경우, JUnit이 테스트 코드를 실행 할 때 Runner class를 직접 명시 할 수 있다. Spring에서 JUnit 4.5+와의 연계를 위해 제공하는 SpringJUnit4ClassRunner로 설정 하여 bean injection등과 같은 Spring이 제공하는 기능을 테스트 코드에서 사용할 수 있게 된다.
- @ContextConfiguration : ApplicationContext 생성 시 필요한 Spring의 bean 속성 정의 파일 위치를 명시 하기 위한 Annotation이다.

JUnit 기반의 테스트 코드 작성 시, 아래의 구문을 이용하여 결과값을 검증할 수 있다.

Name	Description
assertEquals(Object expected, Object actual)	예상되는 결과(expected)와 실제(actual)을 같은지 비교한다.
assertNull(Object obj)	값이 null인 경우 테스트를 통과한다.
assertNotNull(Object obj)	값이 null이 아닌 경우 테스트를 통과한다.
assertSame(Object expected, Object actual)	expected와 actual이 같은 객체를 참조 하는지 확인 하고, 그렇지 않은 경우 실패로 처리한다.
assertNotSame(Object expected, Object actual)	expected와 actual이 다른 객체를 참조 하는지 확인하고, 같은 객체를 참조 하는 경우 실패 처리한다.
assertTrue(boolean condition)	condition이 참(true)인지 확인하고, 그렇지 않은 경우 실패 처리한다.
fail()	이 구문을 만난 경우, 그 즉시 해당 테스트 코드를 실패 처리 한다.

24.2.Test Code 구현

아래의 코드는 JUnit Annotation과 assert 구문을 이용하여 작성한 MovieServiceTest class중 일부이다.

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```

@ContextConfiguration( locations = { "file:./src/main/resources/spring/context-*.xml" })
public class MovieServiceTest {

    @Inject
    @Named("movieService")
    private MovieService movieService;

    @Test
    @Rollback(value = true)
    public void manageMovie() throws Exception {
        // 1. create a new movie
        Movie movie = getMovie();
        movieService.create(movie);

        // 2. assert - create
        movie = movieService.get(movie.getMovieId());
        assertNotNull("fail to fetch a movie", movie);
        assertEquals("fail to compare a movie title", "Shrek (2001)",
            movie.getTitle());

        // 3. update a title of movie
        String title = "Shrek 2 " + System.currentTimeMillis();
        movie.setTitle(title);
        movieService.update(movie);

        // 4. assert - update
        movie = movieService.get(movie.getMovieId());
        assertNotNull("fail to fetch a movie", movie);
        assertEquals("fail to compare a updated title", title, movie.getTitle());

        // 5. remove a movie
        movieService.remove(movie.getMovieId());
    }

    // ...종략
}

```

위 예제 코드에서 @ContextConfiguration annotation을 이용하여 ApplicationContext를 생성한 후, movieService bean을 inject해서 로직을 검증 하는 과정을 확인할 수 있다.

24.3.Test Code 에서의 Meta Annotation 사용

Spring 4.0 에서 추가된 기능으로, 위 Test Code의 @ContextConfiguration 등과 같이 반복되는 Annotation 을 Meta Annotation 으로 등록하여 조금더 간편하게 Test Code에 사용할 수 있다.

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration(locations = { "file:./src/main/resources/spring/context-*.xml" })
public @interface AnyframeTest {}

```

위와 같이 Meta Annotation 을 등록할 수 있으며,

```

@RunWith(SpringJUnit4ClassRunner.class)
@AnyframeTest

```

```
public class MovieServiceMetaAnnotationTest {
    @Inject
    @Named("movieService")
    private MovieService movieService;

    @Inject
    @Named("movieFinder")
    private MovieFinder movieFinder;

    @Test
    @Rollback(value = true)
    public void manageMovie() throws Exception {
        // 1. create a new movie
        Movie movie = getMovie();
        movieService.create(movie);
        //... 중략
    }
}
```

위와 같이 Testcode에 직접 사용할 수 있다.

25.Controller Code Test

이번 절에서는 Controller class 검증을 위한 Testcase 개발 방법에 대해 설명 하고자 한다. 여기서는 Controller class를 테스트 하기 위한 몇 가지 방법 중에서 Controller bean을 직접 Inject 받는 방법을 설명 하고자 한다.

25.1.TestCode

아래는 MovieController class를 테스트하기 위해 작성된 MovieControllerTest 코드의 일부이다. MovieController bean을 @Inject annotation을 이용하여 직접 호출하고 있다.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "file:./src/main/resources/spring/context-*.xml",
    "file:./src/main/resources/spring/*-servlet.xml"})
public class MovieControllerTest {

    private final String SUCCESS_CREATEVIEW = "moviefinder/movie/form";
    private final String SUCCESS_CREATE = "redirect:/movieFinder.do?method=list";
    private final String SUCCESS_GET = "moviefinder/movie/form";
    private final String SUCCESS_UPDATE = "redirect:/movieFinder.do?method=list";

    @Inject
    private MovieController movieController;

    // ...중략

    @Test
    public void testCreate() throws Exception{

        Movie movie = new Movie();
        movie.setTitle("Shrek (2001)");
        movie.setActors("Shrek");
        movie.setDirector("Andrew Adamson");
        Genre genre = new Genre();
        genre.setGenreId("GR-03");
        movie.setGenre(genre);
        movie.setReleaseDate(new Date(20120515));
        movie.setRuntime(new Long(120));
        movie.setTicketPrice(8000f);

        String viewName = movieController.create(movie, new BeanPropertyBindingResult(movie,
            "movie"), new SimpleSessionStatus());

        assertEquals(SUCCESS_CREATE, viewName);

    }

    // ...중략
```

위의 예제 코드에서 @ContextConfiguration annotation을 이용하여 service 및 controller에서 사용하는 bean 설정 등록을 읽어오고 있다. bean으로 등록된 MovieController를 직접 Inject 받아서 테스트 코드 내에서 사용하는 것을 확인할 수 있다.

MovieController가 내부적으로 Service bean을 참조하고 있기 때문에, @ContextConfiguration annotation 등록 시, *-servlet.xml 뿐 아니라 context-*.xml 설정을 함께 등록하는 것을 확인할 수 있다.

26.Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다. 그리고 hsqldb 폴더 내의 start.cmd(or start.sh) 파일을 실행시켜 테스트DB를 시작시켜 놓는다.

- Eclipse 기반 실행

Eclipse에서 압축 해제한 프로젝트를 import 한 후, src/test/java 폴더 하위의 MovieServiceTest, MovieControllerTest, MovieFinderControllerTest를 마우스 우클릭 후 메뉴에서 Run As > JUnit Test를 클릭한다. 그리고 실행 결과를 확인 한다.

Name	Download
hsqldb.test.movie.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/hsqldb.test.movie.zip]
anyframe-sample-test.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-test.zip]

VIII.Message Source

MessageResource Bundle 내에 정의된 텍스트 형태의 정보를 사용자 Locale에 맞게 찾아줌으로써 어플리케이션의 국제화를 지원하기 위해서는 MessageSource를 이용할 수 있다. Anyframe에서는 파일 기반의 MessageResource Bundle과 DB 기반의 MessageResource Bundle 사용을 지원하고 있다. 또한, 정의된 MessageSource가 다수일 경우, 각 MessageSource가 관리하는 여러 MessageResource들로부터 원하는 Message를 찾을 수도 있다.

27.ReloadableResourceBundleMessageSource

웹 애플리케이션을 개발하다보면 MessageSource가 변경될 경우 MessageSource를 반영하기 위하여 애플리케이션 서버를 재시작하는 경우가 있다. MessageSource가 변경될 때마다 서버를 재시작하는 것은 상당히 귀찮은 일이다. 이 같은 문제를 해결하기 위해 Spring 프레임워크에서는 ReloadableResourceBundleMessageSource 클래스를 통해 지원하고 있다. ReloadableResourceBundleMessageSource 클래스의 cacheSeconds 설정을 통해 reloading을 지원한다.

다음은 Core Plugin 설치로 추가된 context-message.xml 파일의 일부로 'messageSource' Bean 정의 내용이다.

```
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>message/message-generation</value>
      <value>message/message-moviefinder</value>
      <value>message/message-converter</value>
    </list>
  </property>
  <property name="defaultEncoding">
    <value>UTF-8</value>
  </property>
</bean>
```

위와 같이 ReloadableResourceBundleMessageSource의 defaultEncoding 속성을 사용하여 Encoding을 정의해 줌으로써 messageSource bean을 정의한다. defaultEncoding을 정의하지 않을 시에는 기본적으로 "UTF-8"로 셋팅된다.



ReloadableResourceBundleMessageSource 사용시 유의할 점

ReloadableResourceBundleMessageSource 클래스를 사용할 때 ResourceBundleMessageSource와 다른 점은 사용할 MessageSource파일을 classpath에 두지 말아야 하는 것이다. 애플리케이션 서버는 클래스패스에 있는 모든 리소스를 캐싱하기 때문에 파일을 변경하더라도 반영되지 않는다. 따라서 ReloadableResourceBundleMessageSource 클래스를 사용하기 위해서는 /WEB-INF/classes 디렉토리 이외의 다른 디렉토리에 MessageSource 파일을 관리해야한다. 예를 들어 /WEB-INF/messages 디렉토리와 같은 곳에서 관리해야한다.

28.DatabaseMessageSource

Spring에서 제공하는 ResourceBundleMessageSource나 이를 확장하여 개발된 Anyframe의 EncodingResourceBundleMessageSource는 모두 파일 기반으로 Message들을 지원한다. Anyframe에서는 대용량의 Message들에 대한 처리 및 보다 안전한 관리가 필요한 경우를 위해 DB 기반의 MessageSource 기능을 제공하고 있다.

이 기능을 제공하는 구현체는 org.anyframe.spring.message.DatabaseMessageSource이다. DatabaseMessageSource는 모든 Locale별 Message들이 1개의 테이블 내에 정의되어 있음을 전제하고 있으므로 Message 관리 테이블은 다음에 해당하는 칼럼들을 포함하도록 구성하고, Locale별 Message를 추가해야 한다. (테이블명 및 칼럼명은 변경 가능하다.)

- KEY 칼럼 : Message Resource 식별을 위한 KEY 정보 보관 (VARCHAR 타입, PRIMARY KEY)
- LANGUAGE 칼럼 : 지정된 Locale에 맞는 Message 추출을 위한 ISO Language Code [<http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>] 정보 보관 (VARCHAR 타입, PRIMARY KEY)
- COUNTRY 칼럼 : 지정된 Locale에 맞는 Message 추출을 위한 ISO Country Code [http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html] 정보 보관 (VARCHAR 타입, PRIMARY KEY)
- TEXT 칼럼 : 추출될 Message 내용 보관 (VARCHAR 타입)

위와 같이 구성된 테이블을 이용하여 DatabaseMessageSource는 입력된 Locale 정보(Language 정보, Country 정보)와 Message Key를 기반으로 적절한 Message를 전달하여 어플리케이션의 국제화를 지원하게 되는 것이다.

28.1.Configuration

DatabaseMessageSource를 활용하기 위해서는 다음과 같은 속성들이 정의되어 있어야 한다. 다음에서 각 속성이 가지는 의미에 대해 알아보기로 하자.

Property	Description	Required	Default Value
dataSource	참조할 dataSource Bean의 id를 정의한다. Message들을 보관하고 있는 DB에 접근하기 위해 필요한 속성이다.	Y	N/A
messageTable	Message들을 관리하는 테이블에 대한 정보를 Properties 형태로 정의한다. Properties 내에는 다음과 같은 Property를 정의할 수 있다. <ul style="list-style-type: none">• table : Message 관리 테이블명 (Default = MESSAGE_SOURCE)• key.column : Message KEY를 저장하기 위한 칼럼명 (Default = KEY)• language.column : 해당 Message가 속한 ISO language code를 저장하기 위한 칼럼명 (Default = LANGUAGE)• country.column : 해당 Message가 속한 ISO country code를 저장하기 위한 칼럼명 (Default = COUNTRY)• text.column : Message 내용을 저장하기 위한 칼럼명 (Default = TEXT)	N	N/A

Property	Description	Required	Default Value
defaultLanguage	DatabaseMessageSource는 defaultLanguage, defaultCountry 속성 정보를 기반으로 기본 locale 정보를 셋팅한다. 그리고 입력된 Locale에 맞는 적절한 Message를 찾지 못할 경우 기본 locale에 맞는 Message를 찾아 전달한다. 만약 기본 locale에 맞는 Message도 존재하지 않을 경우 NoSuchMessageException이 발생한다. 따라서, 기본 적용될 ISO language code [http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt]를 정의한다.	N	en
defaultCountry	DatabaseMessageSource는 defaultLanguage, defaultCountry 속성 정보를 기반으로 기본 locale 정보를 셋팅한다. 그리고 입력된 Locale에 맞는 적절한 Message를 찾지 못할 경우 기본 locale에 맞는 Message를 찾아 전달한다. 만약 기본 locale에 맞는 Message도 존재하지 않을 경우 NoSuchMessageException이 발생한다. 따라서, 기본 적용될 ISO country code [http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html]를 정의한다.	N	US
cacheManager	DatabaseMessageSource는 효율적인 Message 조회를 위해 내부적으로 Ehcache를 사용하고 있다. 한번 조회된 Message는 Cache에 저장되므로, 다음번 조회시 DB에 다시 접근할 필요가 없게 된다. (단, Cache 속성 정의에 따라 달라질 수 있음.) Cache 기능을 연동하려면 Spring에서 제공되는 org.springframework.cache.ehcache.EhCacheCacheManager를 통해 별도 Spring Bean으로 등록된 cacheManager의 id를 정의하여 준다. 기본적으로 Ehcache는 별도 속성 파일을 정의하여 캐싱된 객체를 관리하는데 databaseMessageSourceCache 이름으로 캐시를 추가하여야 한다.	N	N/A
lazyLoad	특정 테이블을 통해 관리되는 모든 Message들을 미리 로드할 것인지 여부를 정의한다. lazyLoad가 true인 경우 getMessage() 메소드 호출을 통해 개별 Message를 찾을 때에 DB에 접근하여 해당 Message를 찾고 내부 Cache에 저장한다. lazyLoad가 false인 경우 DatabaseMessageSource bean이 인스턴스화된 이후, 대상 테이블로부터 모든 Message를 찾아 Cache에 저장한다. 이 경우 cacheConfiguration에 정의된 timeToLive, timeToIdle 속성과 최대 캐싱 데이터 수는 무시된다.	N	true

다음은 위에서 언급한 DatabaseMessageSource의 속성 정의를 포함하고 있는 context-message.xml의 일부이다.

```

<bean id="messageSource"
      class="org.springframework.message.DatabaseMessageSource">
    <property name="dataSource" ref="dataSource"/>
    <property name="messageTable">
        <props>
            <prop key="table">TEST_MESSAGE_SOURCE</prop>
            <prop key="text.column">MESSAGE</prop>
        </props>
    </property>
    <property name="cacheManager" ref="cacheManager" />
    <property name="cacheConfiguration" value="classpath:/spring/message/ehcache.xml"/>
</bean>

```

```
<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager"
    p:cache-manager-ref="ehcache" />

<bean id="ehcache"
    class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
    p:config-location="classpath:spring/message/ehcache.xml" />
```

별도로 정의된 ehcache.xml 파일에는 다음과 같이 databaseMessageSourceCache 이름으로 지정된 캐시를 정의하여 준다. Ehcache 설정을 위한 보다 자세한 내용은 Ehcache 매뉴얼 [<http://ehcache.org/documentation/configuration.html>]을 참고하도록 한다.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../config/ehcache.xsd" name="cacheManager">

    <diskStore path="java.io.tmpdir"/>

    <defaultCache
        maxElementsInMemory="3"
        eternal="false"
        timeToIdleSeconds="1"
        timeToLiveSeconds="1"
        overflowToDisk="true"
        maxElementsOnDisk="10000000"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
    />

    <cache name="databaseMessageSourceCache"
        maxElementsInMemory="3"
        eternal="false"
        timeToIdleSeconds="1"
        timeToLiveSeconds="1"
        overflowToDisk="true"
        maxElementsOnDisk="10000000"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
    />

</ehcache>
```

28.2.Import/Export/Refresh Messages

DatabaseMessageSource는 대량의 Message들을 한번에 Message 관리 테이블에 반영할 수 있도록 하기 위해 Import 기능을 제공한다. 이를 위해서는 DatabaseMessageSource의 importMessages(final List<Message> messages) 메소드를 호출하면 된다. 이 때, 각 Message들은 org.anyframe.spring.message.Message 객체에 담겨져 있어야 함에 유의하도록 한다.

또한 Message 관리 테이블에 보관된 모든 Message들을 추출해낼 수 있도록 하기 위해 Export 기능을 제공한다. 이를 위해서는 DatabaseMessageSource의 List<Message> exportMessages() 메소드를 호출하면 된다. exportMessages() 메소드 호출 결과는 org.anyframe.spring.message.Message 객체의 List 형태로 전달된다. 각 Message 정보는 org.anyframe.spring.message.Message 객체에 담겨진다.

다음은 DatabaseMessageSource를 이용한 Message Import/Export를 수행하는 MessageMgmtServiceImpl의 일부이다.

```
public class MessageMgmtServiceImpl implements MessageMgmtService {
    @Inject
```

```
MessageSource messageSource;

public void importMessages() throws Exception {
    List<Message> messages = new ArrayList<Message>();
    Message message = new Message("error.moviefinderimpl.getpaginglist", "en",
        "US", "Movie List not displayed.");
    messages.add(message);

    // ...
    // add messages to ArrayList

    ((DatabaseMessageSource) messageSource).importMessages(messages);
}

public List<Message> exportMessages() throws Exception {
    return ((DatabaseMessageSource) messageSource).exportMessages();
}
}
```

이 외에 Message 관리 테이블에 변경이 발생하였을 경우 변경 사항을 DatabaseMessageSource 내부의 Cache에 반영하는 작업이 필요할 것이다. 이를 위해서 DatabaseMessageSource는 refresh() 메소드를 제공하고 있으며 호출시 다음과 같이 동작한다. lazyLoad가 true인 경우 DatabaseMessageSource 내부의 캐싱된 데이터가 모두 삭제된다. lazyLoad가 false인 경우 DatabaseMessageSource 내부의 캐싱된 데이터가 모두 삭제되며, Message 관리 테이블로부터 모든 Message 정보를 읽어들이어 Cache에 다시 저장한다.

29. AggregatingMessageSource

어플리케이션 개발시 국제화를 위해 사용될 Locale별 Message 정보에 대해 정보의 성격을 구분하여 각각 다른 유형의 Message Resource에 관리해야 할 수 있다. 이를 위해 Anyframe에서는 `org.anyframe.spring.message.AggregatingMessageSource`를 제공하고 있다. 따라서 다양한 타입의 Message Resource를 사용하여 어플리케이션의 국제화를 지원하고자 하는 경우 `AggregatingMessageSource`를 기본 'messageSource' bean으로 정의하고 `AggregatingMessageSource`를 통해 각 Message Resource에 접근하면 된다.

`AggregatingMessageSource`를 활용하기 위해서는 다음과 같은 속성이 정의되어 있어야 한다. 다음에서 속성이 가지는 의미에 대해 알아보기로 하자.

Property	Description	Required	Default Value
messageSources	Aggregating 대상이 되는 MessageSource bean의 id 목록을 정의한다. 정의되는 bean은 반드시 MessageSource 타입이어야 한다. 이 때 정의된 bean의 순서는 AggregatingMessageSource가 Message를 찾을 때 참조하는 bean의 순서와 동일하다.	Y	N/A

다음은 위에서 언급한 `AggregatingMessageSource`의 속성 정의를 포함하고 있는 `context-message.xml`의 일부이다.

```
<bean name="messageSource"
      class="org.anyframe.spring.message.AggregatingMessageSource">
  <property name="messageSources">
    <list>
      <ref bean="databaseMessageSource"/>
      <ref bean="fileMessageSource"/>
    </list>
  </property>
</bean>

<bean name="fileMessageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="useCodeAsDefaultMessage">
    <value>false</value>
  </property>
  <property name="basenames">
    <list>
      <value>anyframe-message</value>
    </list>
  </property>
</bean>

<bean name="databaseMessageSource"
      class="org.anyframe.spring.message.DatabaseMessageSource">
  <property name="dataSource" ref="dataSource"/>
  <property name="messageTable">
    <props>
      <prop key="table">TEST_MESSAGE_SOURCE</prop>
    </props>
  </property>
  <property name="cacheConfiguration" value="classpath:/spring/message/ehcache.xml"/>
</bean>
```

위의 속성 정의에 의해 `AggregatingMessageSource`는 내부적으로 'databaseMessageSource' bean과 'fileMessageSource' bean을 사용하여 적절한 메시지를 찾게 될 것이다. 즉, `AggregatingMessageSource`

는 입력된 Locale 정보(Language 정보, Country 정보)와 Message Key를 기반으로 먼저 'databaseMessageSource'를 대상으로 메시지를 찾는 작업을 수행하고 적절한 메시지가 존재하지 않으면 'fileMessageSource'를 대상으로 적절한 메시지를 찾는 작업을 다시 시도하게 될 것이다.

IX.Query Service

Query Plugin 매뉴얼 [<http://dev.anyframejava.org/docs/anyframe/plugin/optional/query/1.6.0/reference/htmlsingle/query.html>]을 참조한다.

X.Properties Service

외부 파일이나 환경 정보에 구성되어 있는 key, value의 쌍을 내부적으로 가지고 있으며, 어플리케이션이 이 특정 key에 대한 value에 접근할 수 있도록 해주는 서비스이다. 이 서비스는 주로 시스템의 설치 환경에 관련된 정보나, 잦은 정보의 변경이 요구되는 경우 외부에서 그 정보를 관리하게 함으로써 시스템의 가변성을 향상시킨다. EJB 컴포넌트의 경우는 이미 이러한 정보를 관리할 수 있는 내부적인 기능을 제공하고 있으므로 서비스 내에서는 별도로 이 기능을 사용할 필요는 없다.

다음은 Anyframe 에서 제공하는 Properties 서비스에 대한 구현체이다. (Core Plugin으로 생성된 샘플 어플리케이션에서는 PropertiesService 대신 SpEL을 활용하여 Property 정보를 처리하고 있다. 그러나 Property File에 대한 인코딩 처리 및 Dynamic Reload 기능 등이 필요할 경우 활용할 수 있다.)

30.PropertiesServiceImpl

다음은 Properties 서비스를 사용하기 위해 필요한 설정 정보들이다.

Property Name	Description	Required	Default Value
dynamicReload	PropertiesService를 통해 관리되는 파일들에 대한 변경 여부를 감지하는 주기 (millisecond 단위) 이 속성이 정의되어 있지 않은 경우 Dynamic Reload를 수행하지 않음 (이 속성에 대한 자세한 내용은 본 장의 Dynamic Reloading을 참고하도록 한다.)	N	-1
encoding	property file의 encoding 정보를 정의한다.	N	Empty String
filenames	key, value의 쌍이 외부 별도 파일에 존재하는 경우 해당 파일명을 경로와 함께 표시한다. 절대 / 상대 물리적인 파일 경로 지정 방법과 Classpath를 이용한 지정 방법 2가지가 있다. 둘 이상의 파일의 경우, 콤마(,)로 구분한다.	N	Empty String

30.1.Samples

• Configuration

다음은 Properties 서비스의 속성을 정의한 context-properties.xml 의 일부이다. 아래 PropertiesService는 클래스패스 상에 존재하는 sample-resource.properties 파일에 정의된 property들과 개별 정의된 property들을 관리하게 된다.

```
<bean name="propertiesService"
      class="org.anyframe.util.properties.impl.PropertiesServiceImpl"
      destroy-method="destroy">
  <properties name="dynamicReload" value="1000"/>
  <properties name="encoding" value="UTF-8"/>
  <properties name="filenames">
    <value>
      classpath:/sample-resource.properties
    </value>
  </properties>
</bean>
```

• TestCase

다음은 앞서 정의한 속성 설정을 기반으로 Properties 서비스를 사용하는 MovieDao.java 코드의 일부이다.

```
@Inject
@Named("propertiesService")
PropertiesService propertiesService;

public Page getPagingList(Movie movie){
  // try to paging list based on properties (PAGE_SIZE, PAGE_UNIT)
  int page_unit = propertiesService.getInt("pageUnit");
  int page_size = propertiesService.getInt("pageSize");

  System.out.println("value of PAGE_UNIT property is a '" + page_unit + "'.");
  System.out.println("value of PAGE_SIZE property is a '" + page_size + "'.");
  return new Page();
}
```

다음은 앞서 정의한 속성 설정을 기반으로 Properties 서비스를 사용하여 message 를 추출하는 Main.java 코드의 일부이다.

```
public void getProperty() {  
    // 1. lookup propertiesService  
    PropertiesService service = (PropertiesService) context  
        .getBean("propertiesService");  
    // 2. try to get a property  
    System.out.println("value of message property is a '"  
        + service.getString("message") + "'.");  
}
```

31.Sample Property File

다음은 위 Properties 서비스 속성 정의 파일에 정의된 context.properties 파일의 내용이다.

```
pageSize=3  
pageUnit=10  
  
message=Hi! Anyframe
```

32.Dynamic Reloading

PropertiesService는 관리하는 Property 파일에 대한 Dynamic Reload 기능을 제공한다. (PropertiesService 4.2.0 이후) Dynamic Reload 기능을 사용하게 되면 시스템 운영 중 관리 대상이 되는 Property 파일이 변경되었을 경우 이를 감지하여 내부적으로 Reload를 수행함으로써 서버를 재시작하지 않고서도 변경된 Property 정보를 읽을 수 있게 된다.

Dynamic Reload 기능을 적용하기 위해서는 'dynamicReload' 속성 정의가 추가되어야 한다.



Dynamic Reload 기능 사용시 유의할 점

Dynamic Reload 기능은 Properties Service이 XML 내에 정의된 개별 Property나 클래스패스 내에 존재하는 Property File이 아닌 파일시스템을 통해 로드한 Property File에만 적용됨에 유의하도록 한다. 또한 Dynamic Reload 기능 수행시 이전에 로드되어 있던 Properties를 Clear하고 새로운 Properties를 반영하도록 구현되어 있으므로 PropertiesService Bean 속성 정의시 개별 Property 정의 및 클래스패스 내에 존재하는 Property File은 사용하지 말고 파일 시스템 기반의 Property File만을 활용하도록 한다.

33.Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 anyframe-sample-properties.zip 파일을 다운받은 후, 압축을 해제한다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, `mvn compile exec:java -Dexec.mainClass=...`이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, `src/main/java` 폴더의 `org/anyframe/sample/properties` 하위의 `Main.java`를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 `Run As > Java Application`을 클릭한다. 그리고 실행 결과를 확인한다.

Name	Download
anyframe-sample-properties.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-properties.zip]

XI.Servlet 3.0

서블릿 3.0 은 JSR-315 스펙의 일부로 Java Community Process [<http://jcp.org/aboutJava/communityprocess/final/jsr315/>] 정의되었으며 2.5 버전이후 나온 메이저 버전의 변경이다. 최근의 웹 어플리케이션 개발 스타일을 반영한 여러가지 feature들을 담고 있으며 개발 편의성 향상에 중점을 두고 있다.

서블릿 3.0 스펙의 주요한 feature는 다음과 같다.

- Ease Of Development

서블릿 3.0 스펙에서는 어노테이션 지원, 지너릭 지원등을 통해 개발 편의성을 높일 수 있도록 하였다.

- Pluggability

web.xml 을 모듈화 할 수 있게 해주는 Pluggability 기능이 추가 되었다.

- Asynchronous support

HTTP Request에 대한 비동기 처리 기능이 추가 되었다.

- Security

어노테이션을 이용하여 자원에 대한 접근 제어를 할 수 있는 기능이 추가 되었다.

34. Servlet 3 지원 환경

- JDK

JAVA Platform, standard Edition 6.0 이상

- WAS

Servlet 3.0 이상을 지원하는 WAS 혹은 서블릿 컨테이너

Server	Server version
Apache Tomcat	7.0.x 이상
Jetty	8.x 이상
GlassFish	3.0.x 이상
Weblogic	12.1.x 이상
Websphere	8.0.x 이상
Jeus	7.0.x 이상

- **Servlet 3 WEConfig**

일반적인 Servlet 3 개발 환경 혹은 Servlet 3 표준에 기반한 스프링 3.2.2를 사용하기 위해서는 web.xml 파일을 다음과 같이 수정해야한다.

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="webApp_ID" version="3.0">
```


35.Ease of Development

서블릿 3.0 에 새롭게 추가된 어노테이션과 자바 기반의 설정으로 손쉬운 개발을 할 수 있다.

35.1.Easy configuration through annotations

서블릿 3.0에는 개발자의 편의를 위하여 몇 가지 유용한 어노테이션들이 추가되었다. 개발자는 web.xml 파일에 설정을 추가하는 대신 어노테이션을 이용하여 서블릿, 필터, 리스너 등을 정의할 수 있다. 웹 어플리케이션의 배포서술자는 'metadata-complete' 라는 새로운 속성을 가지는 데 이 속성은 배포시점에 web.xml 파일만으로 배포서술자가 구성되는지 결정하거나 혹은 해당 jar파일의 클래스 파일들이 어노테이션이나 나 web-fragment 파일들을 위해 처리되어야 하는지를 결정한다. 'metadata-complete' 가 'true' 로 정의 된다면 배포시점에서 클래스 파일이나 web-fragment 파일에 선언된 서블릿 어노테이션들은 무시된다. 'metadata-complete' 속성이 정의 되어있지 않거나 'false'로 정의 되어 있다면 배포가 될 때 클래스 파일들과 web-fragment에 에 선언된 어노테이션들이 처리된다.

Servlet 3.0에는 다음과 같은 어노테이션들이 추가되었다.

- **Annotations**

설 명
@WebServlet
@WebInitParam
@WebListener
@WebFilter
@MultipartConfig

35.1.1.@WebServlet

@WebServlet 어노테이션은 웹 어플리케이션 안의 서블릿을 정의할 때 사용 된다. 이 어노테이션은 어떤 클래스에 대하여 정의되며 어노테이션으로 선언되는 서블릿에 대한 메타 데이터를 포함한다. 어노테이션으로 서블릿을 선언할 때는 'urlPatterns' 혹은 'value' 속성이 이 반드시 정의되어야 하는데 그 외의 다른 모든 속성들은 선택적으로 사용할 수 있다. 어노테이션을 통해서 url 만 표기를 할 때에는 value 속성을 사용하는 것을 권장한다. value 속성과 urlPatterns를 동시에 사용할 수 없다. 또한 해당 서블릿이 서블릿 컨테이너에 배포가 되려면 어노테이션을 사용한 서블릿은 적어도 하나의 url 패턴을 정의해야 한다. 서블릿의 이름을 따로 정의를 하지 않는다면 패키지 경로를 포함한 클래스 명이 서블릿의 이름이 된다. @WebServlet 어노테이션을 사용하는 클래스는 반드시 javax.servlet.http.HttpServlet 클래스를 상속해야 한다. @WebServlet으로 어떤 서블릿을 선언하였을 때 배포서술자 web.xml에 같은 name 속성으로 선언된 동일한 서블릿이 있다면 배포서술자에 선언된 서블릿이 어노테이션으로 선언한 서블릿을 오버라이드 한다는 점에 주의해야 한다. (어노테이션으로 선언된 서블릿의 인스턴스는 초기화된다) 배포서술자에 선언된 서블릿과 어노테이션 서블릿의 이름이 일치하지 않는다면 각각 다른 서블릿이 생성된다.

다음은 @WebServlet을 사용한 예제이다.

```
package org.anyframe.sample.servletannotation.web.servlet;
... (생략) ...
@WebServlet("/movieFinder.do")
public class MovieFinderServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    static Logger logger = Logger.getLogger(MovieFinderServlet.class);
```

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    logger.info("[ " + getServletName() + " ]" + " Servlet works.");
    .. 생략...
    RequestDispatcher rd = request
        .getRequestDispatcher("/WEB-INF/jsp/moviefinder/movie/list.jsp");
    rd.forward(request, response);
}
```

위의 코드를 보면 @WebServlet 어노테이션을 통해 간단히 서블릿을 선언하고 있는 것을 알 수 있다. 위와 같이 아무 속성을 정의하지 않고 문자열만 입력하면 입력된 값이 url로 간주되며 서블릿 명은 패키지 경로를 포함한 'org.anyframe.sample.servletannotation.web.servlet.MovieFinderServlet'이 된다. 서블릿 명을 임의로 지정하고 싶다면 'name' 속성을 정의하고 서블릿을 호출할 url을 'urlPatterns'로 입력 받으면 된다. 위의 서블릿을 서블릿 2.5에서 web.xml을 이용하여 설정을 하면 다음과 같을 것이다.

위에서 언급한 payload 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-servlet-annotation 를 통해 다운로드받을 수 있다.

```
<servlet>
  <servlet-name>
    org.anyframe.sample.servletannotation.web.servlet.MovieFinderServlet
  </servlet-name>
  <servlet-class>
    org.anyframe.sample.servletannotation.web.servlet.MovieFinderServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>
    org.anyframe.sample.servletannotation.web.servlet.MovieFinderServlet
  </servlet-name>
  <url-pattern>/movieFinder.do</url-pattern>
</servlet-mapping>
```

위의 xml 예제를 보면 어노테이션으로 선언한 서블릿과 동일한 url을 사용하고 있다. 만약 서블릿 3.0 API를 통해서 개발자가 어노테이션으로 서블릿을 선언한 상태에서 위와 같은 설정을 xml에 추가한다면 'java.util.concurrent.ExecutionException: org.apache.catalina.LifecycleException' 같은 오류가 발생할 것이다. 테스트를 한 환경은 Tomcat 7.0버전이다. 개발자는 어노테이션과 xml을 동시에 사용할 경우에 주의를 해서 사용할 필요가 있다.

35.1.2. @WebInitParam

@WebInitParam 어노테이션은 단독으로는 사용할 수 없고 어노테이션을 통해서 서블릿이나 필터를 등록할 때 속성을 정의하는데 사용되는 어노테이션이다. 예제는 @WebServlet 어노테이션과 함께 @WebInitParam 어노테이션을 사용한 예제이다. 매우 간단한 방법으로 사용이 가능함을 알 수 있다.

```
@WebServlet(name = " FileUploadForwardServlet ", urlPatterns = "/fileUpload.do", initParams =
{
    @WebInitParam(name = "encoding", value = "utf-8"),
    @WebInitParam(name = "method", value = "list") })
public class FileUploadForwardServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    static Log log = LoggerFactory.getLog(FileUploadForwardServlet.class);

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        log.info("[ " + getServletName() + " ]" + " Servlet works.");
        log.info("Encoding type : [ " + getInitParameter("encoding") + " ]");
    }
}
```

```
...(생략)...
response.sendRedirect("http://localhost:8080/anyframe-sample-eod/anyframe.jsp");
}
}
```

위에서 언급한 payload 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-servlet-annotation 를 통해 다운로드받을 수 있다.

35.1.3.@WebFilter

@WebFilter 어노테이션은 웹 어플리케이션을 위한 서블릿 필터를 정의한다. 필터의 이름을 정의하지 않는다면 패키지를 포함한 클래스 명이 필터의 이름으로 사용된다. urlPatterns 속성, servletNames 속성 혹은 value속성이 반드시 정의 되어야 하며 나머지 속성들은 선택적으로 사용할 수 있다. @WebServlet 어노테이션과 마찬가지로 web.xml파일에 선언된 filter와 이름이 같다면 배포서술자는 web.xml파일에 선언된 filter에 의해서 정의된다. 필터의 이름이 같지 않다면 각각 다른 필터가 생성될 것이다. 'asyncSupported' 속성을 true로 설정하여 비 동기 서블릿을 위한 비 동기 필터로 선언할 수 있다. @WebFilter어노테이션을 사용하는 클래스들은 반드시 javax.servlet.Filter 인터페이스를 구현해야 한다.



참고

비 동기 서블릿에 사용하는 필터를 동기로 선언하면 실행 시에 'illegalStateException'이 발생하니 사용에 주의해야 한다. 또한 @WebServlet 어노테이션과 마찬가지로 urlPatterns만 사용한다면 value 속성만 정의하는 것을 권장하고 다른 속성들을 사용한다면 urlPatterns 속성을 사용하는 것이 좋다.

다음은 @WebListener를 통해서 필터를 선언한 예제 코드이다.

```
@WebFilter("/movie.do")
public class EncodingFilter implements Filter {

    public void destroy() {
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        request.setAttribute("encoding", "utf-8");
        chain.doFilter(request, response);
    }
    public void init(FilterConfig fConfig) throws ServletException {
    }
}
```

위에서 언급한 payload 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-servlet-annotation 를 통해 다운로드받을 수 있다.

35.1.4.@WebListener

서블릿 컨테이너 리스너는 언제라도 서블릿 컨텍스트가 생성되는 이벤트를 받기 위하여 사용되고 웹 컨테이너에 의해서 종료된다. @WebListener 어노테이션을 사용하기 위해서는 다음의 클래스들 중에 하나를 구현해야 한다.

Resource Implementation	Purpose
javax.servlet.ServletContextListener	서블릿 컨텍스트의 라이프 사이클의 상태 변화에 관련된 이벤트 알림을 받는 인터페이스

Resource Implementation	Purpose
javax.servlet.ServletContextAttributeListener	서블릿 컨텍스트 속성의 변화에 관련된 이벤트 알림을 받는 인터페이스
javax.servlet.ServletRequestListener	웹 어플리케이션 스코프로 요청이 입출력되는 이벤트 알림을 받는 인터페이스
javax.servlet.ServletRequestAttributeListener	서블릿 요청 속성 변화에 대한 이벤트 알림을 받는 인터페이스
javax.servlet.http.HttpSessionListener	HTTP 세션 라이프 사이클의 상태변화에 관련된 이벤트를 받는 인터페이스
javax.servlet.http.HttpSessionAttributeListener	HTTP 세션 속성의 변화에 관련된 이벤트 알림을 받는 인터페이스

다음은 @WebListener 어노테이션을 통해서 리스너를 구현한 예제이다

```
@WebListener
public class AnyframeServletContextListener implements ServletContextListener {

    static Logger logger = Logger.getLogger(AnyframeServletContextListener.class);

    public void contextInitialized(ServletContextEvent sce) {
        logger.info("[AnyframeServletContextListener] Servlet Context has been initialized");
    }
    public void contextDestroyed(ServletContextEvent sce) {
        // TODO Auto-generated method stub
    }
}
```

위의 예제코드에서는 @WebListener annotation을 통해서 Servlet Context의 생성과 소멸과 관련된 알림을 받을 수 있는 리스너를 구현하고 있다. ServletContextListener 클래스에서 제공하는 contextInitialized 클래스를 구현함으로써 Servlet Context가 초기화 되는 시점에 프로그래밍적인 방법으로 서블릿, 필터, 리스너 등을 추가할 수 있다. 어노테이션을 통해서 선언할 수 있는 다른 클래스의 사용법은 클래스의 이름으로 쉽게 유추할 수 있다.

위에서 언급한 payload 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-servlet-annotation 를 통해 다운로드받을 수 있다.

35.1.5.@MultipartConfig

서블릿 3.0에서는 @MultipartConfig 어노테이션을 통해서 파일 업로드 기능을 제공한다. 이 어노테이션을 서블릿에 사용하면 해당 서블릿이 기대하는 요청이 mime/multipart 임을 가르킨다. 서블릿의 HttpServlet 객체는 반드시 다양한 mime 첨부 파일들을 반복하여 처리할 수 있는 getParts와 getPart 메소드를 통하여 mime첨부 파일을 사용할 수 있도록 해야 한다. getPart를 통해서 얻는 객체는 Spring 에서 제공하는 MultiPartFile과 그 용도가 비슷한 Wrapper Object이다. 아래에서 제공하는 html 예제는 파일 업로드를 위하여 서블릿을 만든 폼 이다. 해당하는 파일을 이용하여 파일업로드를 사용하기 위하여는 다음과 같은 조건을 만족해야한다

enctype 속성은 반드시 **multipart/form-data**. 값으로 설정되어야 한다.

해당 폼 파일을 사용하기 위한 서블릿의 메소드는 반드시 **post** 방식이어야한다.

해당 form이 이렇게 정의가 되면 요청 전체가 multipart/form-data 타입으로 서버로 넘겨진다. 그러면 서블릿이 해당 입력 스트림으로부터 파일 데이터를 처리하여 파일을 추출하기 위하여 요청을 처리한다. destination은 컴퓨터에 파일이 저장될 위치를 지정하는 경로이다. form 하단의 Upload 버튼을 누르면 지정한 경로에 파일을 저장하는 서블릿으로 데이터를 보낸다. HTML form 파일은 다음과 같다.

```
<form method="POST" action="upload" enctype="multipart/form-data" >
    File:
    <input type="file" name="file" id="file" /> <br/>
    Destination:
    <input type="text" value="c:/tmp" name="destination"/>
    </br>
    <input type="submit" value="Upload" name="upload" id="upload" />
</form>
```

POST 방식의 요청 메소드는 어떤 파일을 업로드 할 때나 온전한 form 을 제출하는 것처럼 클라이언트가 데이터를 요청을 일부분으로 서버로 보낼 때 사용된다. GET 방식의 요청 메소드는 URL과 헤더정보만 서버로 보내는 반면에 POST 방식의 요청들은 메시지 바디 자체를 포함한다. 이로 인해 길이를 일정하지 않거나 매우 긴 어떤 타입의 데이터도 서버로 보낼 수 있다. POST 요청 메소드의 헤더 필드는 보통 메시지 바디의 미디어 타입을 나타낸다. 다음은 파일 업로드를 위한 POST 메소드를 구현한 예제코드이다.

```
public class FileUploadServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    static Logger logger = LoggerFactory.getLogger(FileUploadServlet.class);

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {

        response.setContentType("text/html;charset=UTF-8");

        // Create path components to save the file
        final String path = request.getParameter("destination");
        final Part filePart = request.getPart("file");
        final String fileName = getFileName(filePart);

        OutputStream outputStream = null;
        InputStream inputStream = null;
        final PrintWriter printWriter = response.getWriter();

        try {
            outputStream = new FileOutputStream(new File(path + File.separator
                + fileName));
            inputStream = filePart.getInputStream();

            int read = 0;
            final byte[] bytes = new byte[1024];

            while ((read = inputStream.read(bytes)) != -1) {
                outputStream.write(bytes, 0, read);
            }
            printWriter.println("New file " + fileName + " created at " + path);
            logger.info("File" + fileName + "being uploaded to " + path);
        } catch (FileNotFoundException fne) {
            printWriter.println("You either did not specify a file to upload or are "
                + "trying to upload a file to a protected or nonexistent "
                + "location.");
            printWriter.println("<br/> ERROR: " + fne.getMessage());

            logger.info("Problems during file upload. Error : "
                + fne.getMessage());
        } finally {
            if (outputStream != null) {
                outputStream.close();
            }
            if (inputStream != null) {
```

```

        inputStream.close();
    }
    if (printWriter != null) {
        printWriter.close();
    }
}

private String getFileName(final Part part) {
    final String partHeader = part.getHeader("content-disposition");
    logger.info("Part Header = " + partHeader);
    for (String content : part.getHeader("content-disposition").split(";")) {
        if (content.trim().startsWith("filename=")) {
            return content.substring(content.indexOf('=') + 1).trim()
                .replace("\"", "");
        }
    }
    return null;
}
}

```

doPost 메소드는 요청으로부터 destination 속성과 file part 객체를 가지고 온 다음 file part 객체로부터 파일명을 가지고 오기 위하여 getFileName() 메소드를 호출한다. getFileName()가 호출되면 FileOutputStream 객체를 생성하고 지정한 위치에 해당 파일을 복사한다.

위에서 언급한 payload 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-servlet-annotation 를 통해 다운로드받을 수 있다.

35.2. Programmatic Configuration

서블릿 3.0에서는 서블릿, 리스너, 필터 등을 등록하기 위하여 어노테이션과 web.xml을 제공하는 것 외에 프로그래밍 적인 방법을 제공한다. 각각의 서블릿, 리스너, 필터를 등록하기 위하여 다양한 설정 메소드 들을 추가하였다. 아래의 예제에서는 어노테이션으로 선언한 리스너에 프로그래밍적인 방법으로 필터와 서블릿을 등록하는 법을 보여주고 있다. ServletContext 클래스에 추가된 addFilter 메소드는 인자로 전달받은 필터의 인스턴스를 함께 전달받은 필터명으로 서블릿 컨텍스트에 등록한다. 해당 메소드는 javax.servlet.Registration 을 상속받은 FilterRegistration.Dynamic 인터페이스의 객체를 리턴한다. 마찬가지로 addServlet 메소드는 전달받은 서블릿의 인스턴스를 함께 전달받은 서블릿 명으로 서블릿 컨텍스트에 등록한다. 해당 메소드는 javax.servlet.Registration을 상속받은 ServletRegistration.Dynamic 인터페이스의 객체를 리턴한다.

```

@WebListener
public class AnyframeServletContextListener implements ServletContextListener {

    static Logger logger = LoggerFactory
        .getLogger(AnyframeServletContextListener.class);

    public void contextInitialized(ServletContextEvent sce) {
        logger.info("[AnyframeServletContextListener] Servlet Context has been initialized");

        ServletContext sc = (ServletContext) sce.getServletContext();

        // 1. add a filter which filters request with URL '/movie.do'.
        // ServletContext#addFilter method returns FilterRegistration.Dynamic
        // object. Users can configure filter by method which implements
        // FilterRegistration.Dynamic interface
        FilterRegistration.Dynamic dynamicFilter = sc
            .addFilter(

```

```

        "EncodingFilter",
        org.anyframe.sample.servlet.javaconfig.web.filter.EncodingFilter.class);
dynamicFilter.addMappingForUrlPatterns(
    EnumSet.allOf(DispatcherType.class), true, "/movie.do");
logger.info("Encoding Filter has been Added");

// 2. add 'MovieFinderServlet' servlet. It is similar to adding a filter
// that ServletContext#addServlet method
// returns ServletRegistration.Dynamic object
ServletRegistration.Dynamic dynamicServlet = sc
    .addServlet(
        "MovieFinderServlet",
        (org.anyframe.sample.servlet.javaconfig.web.servlet.MovieFinderServlet.class));
dynamicServlet.addMapping("/movieFinder.do");
logger.info("MovieFinder Servlet has been Added");

// 3. add 'MovieServlet' servlet.
dynamicServlet = sc
    .addServlet(
        "MovieServlet",
        (org.anyframe.sample.servlet.javaconfig.web.servlet.MovieServlet.class));
dynamicServlet.addMapping("/movie.do");
logger.info("Movie Servlet has been Added");

// 4. add 'FileUploadForwardServlet' servlet.
dynamicServlet = sc
    .addServlet(
        "FileUploadForwardServlet",
        (org.anyframe.sample.servlet.javaconfig.web.servlet.FileUploadPageServlet.class));
dynamicServlet.addMapping("/fileUpload.do");
dynamicServlet.setInitParameter("encoding", "utf-8");
logger.info("FileUploadForward Servlet has been Added");

// 5. add 'FileUploadForwardServlet' servlet.
dynamicServlet = sc
    .addServlet(
        "FileUploadServlet",
        (org.anyframe.sample.servlet.javaconfig.web.servlet.FileUploadServlet.class));
dynamicServlet.addMapping("/upload");
dynamicServlet.setMultipartConfig(new MultipartConfigElement(null,
    500000000, 500000000, 0));

logger.info("FileUpload Servlet has been Added");

// 6. get FileUploadServlet servlet from Servlet Context through
// ServletContext#getServletRegistration. It returns
// ServletRegistration.Dynamic object so that users can set
// configuration
dynamicServlet = (Dynamic) sc
    .getServletRegistration("FileUploadServlet");
dynamicServlet.setInitParameter("encoding", "utf-8");
logger.info("Add an init parameter on FileUploadServlet Servlet");
}

public void contextDestroyed(ServletContextEvent sce) {
    // TODO Auto-generated method stub
}
}

```

위에서 언급한 payload 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-servlet-annotation 를 통해 다운로드받을 수 있다.

이미 어노테이션을 통해서 해당 필터와 서블릿을 등록하였다면 프로그래밍적인 방법으로 필터와 서블릿을 추가적으로 등록할 필요는 없다. 만약 web.xml 이나 어노테이션을 통해서 등록한 서블릿과 동일한 이름의 서블릿을 (클래스이름이 아닌 서블릿 명) 서블릿 컨텍스트에서는 web.xml > 어노테이션 > 프로그래밍적인 방법의 순으로 서블릿을 생성한다. 아래의 예러코드는 어노테이션으로 "MovieServlet"을 등록하고 프로그래밍 적인 방법으로 동일한 이름의 서블릿을 등록할 때 나는 예러다.



참고

심각: Exception sending context initialized event to listener instance of class org.anyframe.sample.servletjavaconfig.web.listener.AnyframeServletContextListener
java.lang.NullPointerException

Servlet Context가 초기화 될 때 아래와 같은 예제코드와 같은 형태로 서블릿을 등록하였는데 이미 어노테이션으로 'MovieServlet' 이란 서블릿이 등록이 되었기 때문에 Servlet Context의 addServlet은 null값을 리턴한다. 그리하여 addMapping 메소드를 통하여 url패턴을 추가하려고 하면 예러가 발생하게 된다.

```
...
dynamicServlet = sc.addServlet("MovieServlet",
(org.anyframe.sample.servletjavaconfig.web.servlet.MovieServlet.class));
dynamicServlet.addMapping("/movie.do");
...
```

필터와 리스너의 경우에도 서블릿의 경우와 같다. 특히 어노테이션을 통해서 서블릿, 필터, 리스너를 등록할 때에 name 속성을 정의하지 않으면 패키지 경로를 포함한 클래스 명이 서블릿 명이 되기 때문에 해당하는 방법으로 설정을 할 때에는 각 요소의 이름을 지정하는데 유의해야 한다. 이런 혼란을 피하기 위하여 개발자는 어노테이션이나 web.xml 배포서술자를 통하여 서블릿을 선언하는 경우와 프로그래밍적인 방법으로 서블릿을 동적으로 추가하는 경우의 사용을 구분하여 사용할 필요가 있다.

35.3.Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다. 그리고 hsqldb 폴더 내의 start.cmd (or start.sh) 파일을 실행시켜 테스트 DB를 시작시켜 놓는다

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭 하고 컨텍스트 메뉴에서 Maven > Enable Dependency Management를 선택하여 컴파일 에러를 해결한다. 그리고 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭한 후, 컨텍스트 메뉴에서 Run As > Run on Server (Tomcat 기반)를 클릭한다. Tomcat Server가 정상적으로 시작되었으면 브라우저를 열고 주소창에 <http://localhost:8080/anyframe-sample-servlet-annotation> (혹은 <http://localhost:8080/anyframe-sample-servlet-javaconfig>) 를 입력하여 실행 결과를 확인한다.

Name	Download
anyframe-sample-servlet-annotation.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-servlet-annotation.zip]
anyframe-sample-servlet-javaconfig.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-servlet-javaconfig.zip]

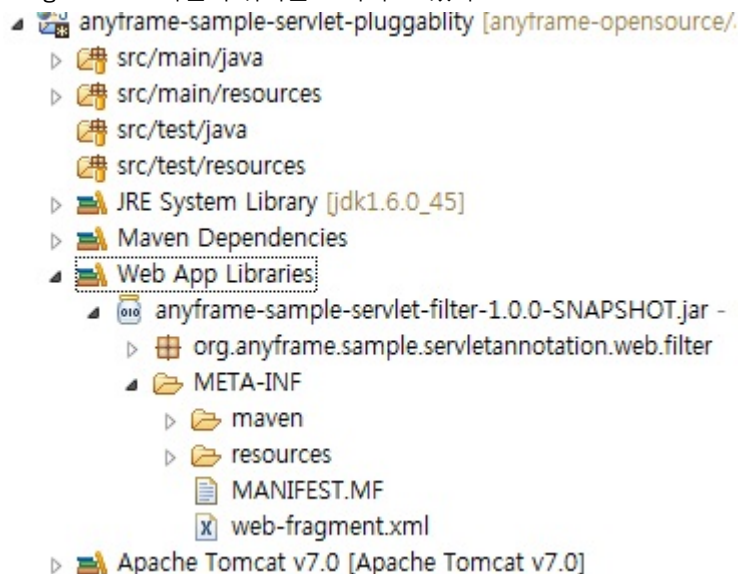
36.Pluggability

Pluggability는 web fragment를 이용한 web.xml의 모듈화, 정적인 자원에 대한 공유 기능 등을 제공하는 Servlet 3.0에서 지원하는 새로운 기능이다.

36.1.Web Fragment

프레임워크(혹은 라이브러리, 이하 프레임워크)를 사용하기 위해서 web.xml(웹 어플리케이션 배포 기술자) 파일에 해당 프레임워크에 대한 설정을 추가 해야 하는 경우가 있다. Web Fragment는 이러한 프레임워크에 대한 설정을 web.xml 파일에서 분리 시켜 모듈화 하고 프레임워크에 대한 설정은 해당 프레임워크 내에 설정 할 수 있도록 해준다. web-fragment.xml은 프레임워크에 대한 descriptor 파일이고 해당 프레임워크의 jar/META-INF 디렉토리에 위치 한다.

다음 그림은 web-fragment.xml 파일의 위치를 보여 주고 있다.



web-fragment.xml 파일의 설정 예제는 다음과 같다.

```
<web-fragment>
<name>Logging-Filter</name>
<filter>
  <filter-name>loggingFilter</filter-name>
  <filter-class>org.anyframe.sample.servletannotation.web.filter.LoggingFilter</filter-
class>
</filter>
<filter-mapping>
  <filter-name>loggingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
</web-fragment>
```

하나의 어플리케이션에서 다수의 web-fragment.xml 이 있는 경우 각각의 web-fragment.xml 파일에 대한 구분은 <name> 태그를 이용한다. 여러개의 web-fragment가 사용되는 경우 적용되는 순서(ex. 서블릿 적용 우선 순위 등)를 지정 할 수 있다. web.xml 파일에 지정하는 <absolute-ordering> 방식(절대적인 순서 지정)과 web-fragment.xml 파일에 지정하는 <ordering> 방식(상대적인 순서 지정) 방식이 있다. 우선 순위를 지정하는 방식에 대한 사용 예제는 다음과 같다.

- **absolute-ordering**

```
<web-app>
  <absolute-ordering>
    <name>MyFragment3</name>
    <name>MyFragment2</name>
  </absolute-ordering>
  ...
</web-app>
```

- **ordering**

```
<web-fragment>
  <name>MyFragment1</name>
  <ordering>
    <after>
      <name>MyFragment2</name>
    </after>
  </ordering>
  ...
</web-fragment>
```

```
<web-fragment>
  <name>MyFragment2</name>
  ..
</web-fragment>
```

```
<web-fragment>
  <name>MyFragment3</name>
  <ordering>
    <before>
      <others/>
    </before>
  </ordering>
  ...
</web-fragment>
```

위와 같이 상대적인 순서를 지정 하는 경우 해당 fragment의 적용 순서는 MyFragment3, MyFragment2, MyFragment1 이 된다.

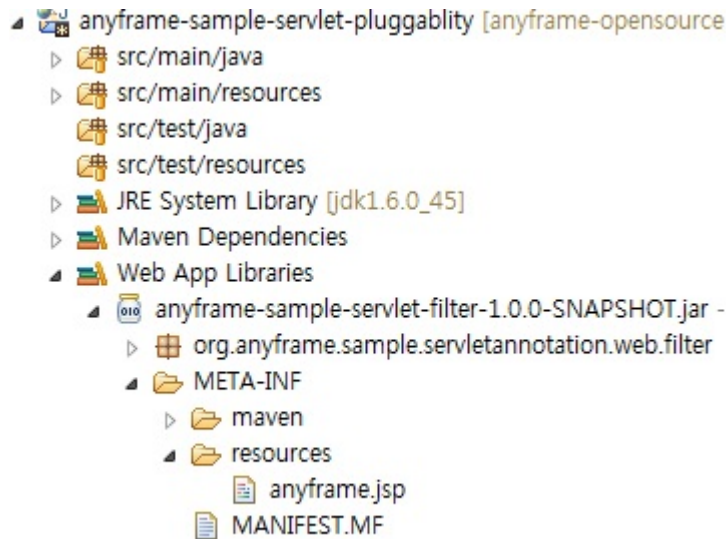


참고

web.xml 파일의 root 태그인 <web-app> 태그의 속성인 metadata-complete 속성을 정의 함으로 써 web-fragment 사용 여부를 결정 할 수 있다. 해당 속성을 정의 하지 않거나 false 로 설정 하는 경우 web-fragment를 사용하게 되고 true인 경우는 web-fragment를 사용하지 않는다.

36.2.Resource Sharing

웹 어플리케이션에서 사용되는 정적인 자원(html, css, js, images, 정적인 일부 jsp 등)들을 document root에 위치 시키지 않고 jar 파일로 관리가 가능하다. 이 경우 jar 파일은 WEB-INF/lib 에 위치 하게 되며 정적인 자원들은 WEB-INF/lib/[*.jar]/META-INF/resources 하위로 위치하게 된다. ServletContext 인터페이스에서 제공하는 getResource, getResourceAsStream 메소드를 통해서 해당 자원들을 접근 할 수 있다. 정적인 자원들을 jar 파일로 관리하게 되면 다수의 프로젝트에서 정적인 자원에 대한 공유가 필요 한 경우 등에서 유용하게 사용을 할 수 있다.



위와 같이 정적인 자원들이 반드시 document root에 위치 할 필요가 없는 것이다. 위와 같이 사용을 하는 경우 anyframe.jsp 파일은 `http://localhost:8080/anyframe-sample-servlet-pluggability/anyframe.jsp` 로 접근이 가능하다. 단, Resource Sharing의 경우는 정적인 자원(html, css, js, images, 정적인 일부 jsp 등)에 대해서만 처리가 가능하다. 동적으로 처리되어야 하는 jsp와 같은 경우(정적인 jsp는 사용 가능)는 Resource Sharing 을 사용 할 수 없다.

36.3.Resources

- 다운로드

다음의 예제는 Pluggability 에 소개한 내용으로 샘플 코드를 작성한 것이다. 별도로 배포하는 필터에 대한 설정을 프로젝트내의 anyframe-sample-servlet-filter-1.6.0.jar 파일안에 설정함으로서 Servlet 3 의 특징인 pluggability 를 보여주고 있다. 실행 방법은 다음과 같다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭하고 컨텍스트 메뉴에서 Maven > Enable Dependency Management를 선택하여 컴파일 에러를 해결한다. 그리고 해당 프로젝트에 대해 마우스 오른쪽 버튼을 클릭한 후, 컨텍스트 메뉴에서 Run As > Run on Server (Tomcat 기반)를 클릭한다. Tomcat Server가 정상적으로 시작되었으면 브라우저를 열고 주소창에 `http://localhost:8080/anyframe-sample-servlet-pluggability/` (포트는 사용자가 정의한 포트) 를 입력하여 실행 결과를 확인한다.

Name	Download
anyframe-sample-servlet-filter.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-servlet-filter.zip]
anyframe-sample-servlet-pluggability.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/sample/anyframe-sample-servlet-pluggability.zip]

37. Asynchronous Support

Async Support Plugin 매뉴얼 [<http://dev.anyframejava.org/docs/anyframe/plugin/optional/async-support/1.1.0/reference/htmlsingle/async-support.html>]을 참조한다.

38.Security Enhancement

서블릿 3.0에 추가된 어노테이션을 사용하여 보안과 관련된 설정을 할 수 있다.

- **@ServletSecurity**

서블릿 클래스에서 사용되며 서블릿 컨테이너에서 HTTP 프로토콜을 통해 주고 받는 메시지에 적용되는 보안 제약 사항을 정의 할 수 있다. 서블릿 클래스에 정의 할 수 있으며 메소드 레벨에는 정의 할 수 없다.

Element	Description	Default
value	httpMethodConstraints에 표현 되지 않은 모든 HTTP 메소드에 대한 보안 제약 사항	@HttpConstraint
httpMethodConstraints	특정 HTTP 메소드에 대한 보안 제약 사항	{}

- **@HttpConstraint**

모든 HTTP 메소드에 적용 되는 보안 제약 사항을 정의 할 수 있으며 @ServletSecurity 하위 요소로 사용되며 HttpMethodConstraint 요소 내에 정의 되지 않는다.

Element	Description	Default
value	rolesAllowed 항목이 빈 값인 경우 적용 되는 기본 보안 제약 사항	PERMIT
rolesAllowed	접근 권한을 가지는 role name	{}
transportGuarantee	전송되는 데이터에 대한 보안 제약 사항	NONE

- **@HttpMethodConstraint**

특정 HTTP 메소드에 적용 되는 보안 제약 사항을 정의 할 수 있으며 @ServletSecurity 하위 요소로 사용된다.

Element	Description	Default
emptyRoleSemantic	rolesAllowed 항목이 빈 값인 경우 적용 되는 기본 보안 제약 사항	PERMIT
rolesAllowed	접근 권한을 가지는 role name	{}
transportGuarantee	전송되는 데이터에 대한 보안 제약 사항	NONE

접근제어 어노테이션을 사용한 예제는 다음과 같다.

- **Code Example 1. 보안 제약 사항 없음**

```
public class Example1 extends HttpServlet {  
}
```

- **Code Example 2. 모든 요청에 대해 보안 제약 사항 없음. 데이터 전송에 대한 보안 필요**

```
@ServletSecurity(@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL))
```

```
public class Example2 extends HttpServlet {  
}
```

- **Code Example 3.** 모든 요청을 거부

```
@ServletSecurity(@HttpConstraint(EmptyRoleSemantic.DENY))  
public class Example3 extends HttpServlet {  
}
```

- **Code Example 4.** 모든 요청에 대해 **R1 role**을 가진 사용자만 접근 허용

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "R1"))  
public class Example4 extends HttpServlet {  
}
```

- **Code Example 5.** **GET, POST**를 제외한 요청에 대한 보안 제약 없음. **GET, POST** 요청은 **R1 role**을 가진 사용자만 접근 허용. **POST** 요청은 데이터 전송에 대한 보안 필요

```
@ServletSecurity((httpMethodConstraints = {  
    @HttpMethodConstraint(value = "GET", rolesAllowed = "R1"),  
    @HttpMethodConstraint(value = "POST", rolesAllowed = "R1",  
        transportGuarantee = TransportGuarantee.CONFIDENTIAL)  
}))  
public class Example5 extends HttpServlet {  
}
```

- **Code Example 6.** **GET** 요청에 대한 보안 제약 사항 없음. **GET**을 제외한 요청에 대해서는 **R1 role**을 가진 사용자만 접근 허용

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),  
    httpMethodConstraints = @HttpMethodConstraint("GET"))  
public class Example6 extends HttpServlet {  
}
```

- **Code Example 7.** **TRACE** 요청은 모두 거부. **TRACE**를 제외한 요청은 **R1 role**을 가진 사용자만 접근 허용

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),  
    httpMethodConstraints = @HttpMethodConstraint(value="TRACE",  
        emptyRoleSemantic = EmptyRoleSemantic.DENY))  
public class Example7 extends HttpServlet {  
}
```