

전자정부 표준프레임워크 실행환경 (공통기반)

eGovFrame



Contents

1. _ 실행환경 소개
2. _ 공통기반 레이어



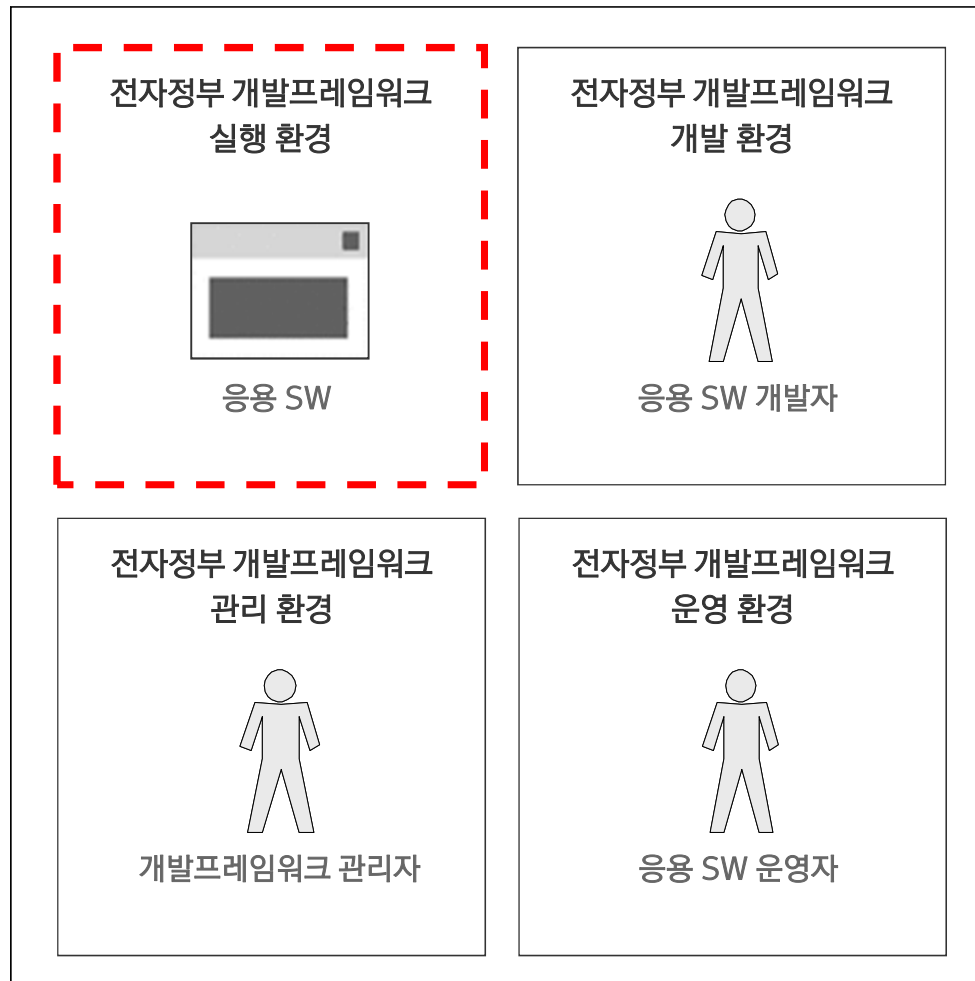


1. 실행환경 소개

1. 개요
2. 배경
3. 실행환경 특징
4. 실행환경 적용효과
5. 실행환경 구성
6. 실행환경 오픈 소스 SW 사용현황

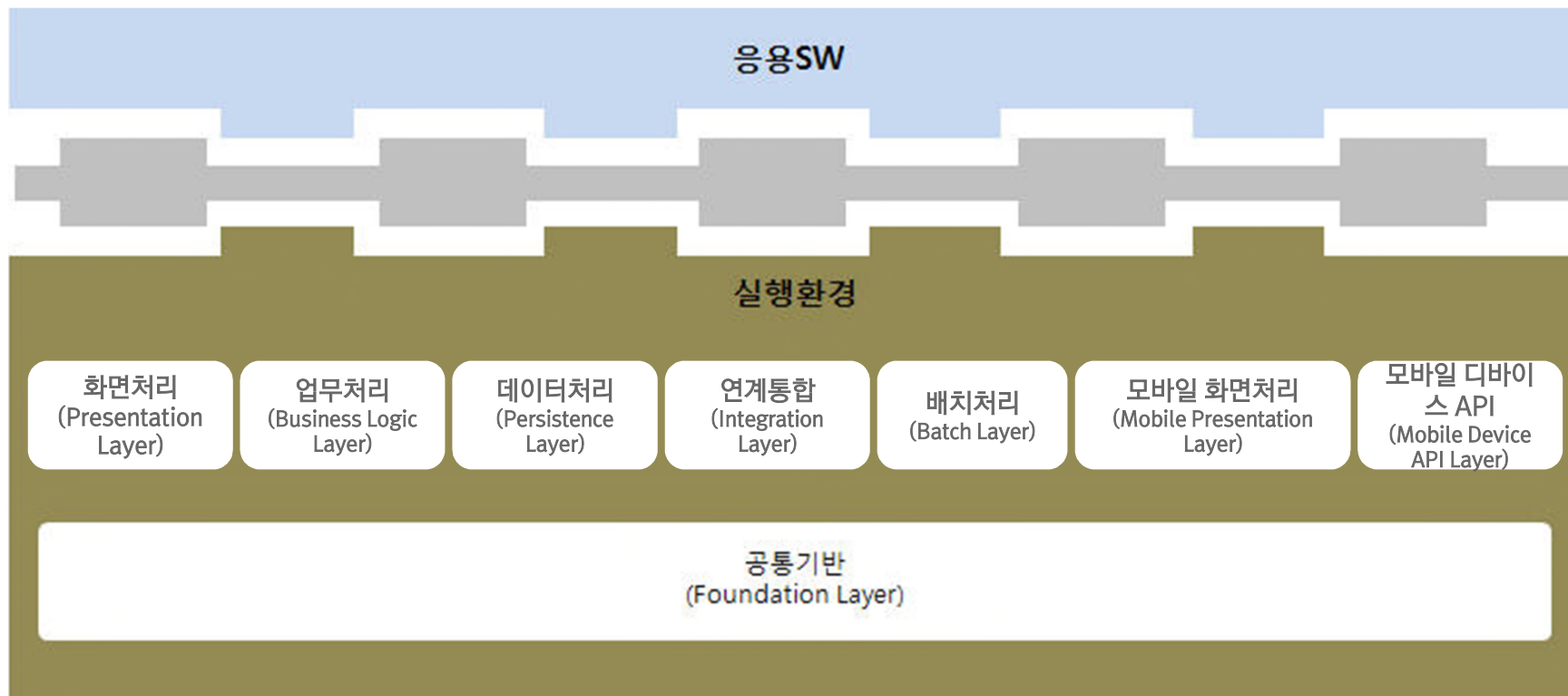
- 전자정부 개발프레임워크 환경은 응용 SW를 위한 실행 환경, 응용 SW 개발자를 위한 개발 환경, 응용 SW 운영자를 위한 운영 환경, 개발프레임워크 관리자를 위한 관리 환경으로 구성됨

전자정부 개발프레임워크 환경

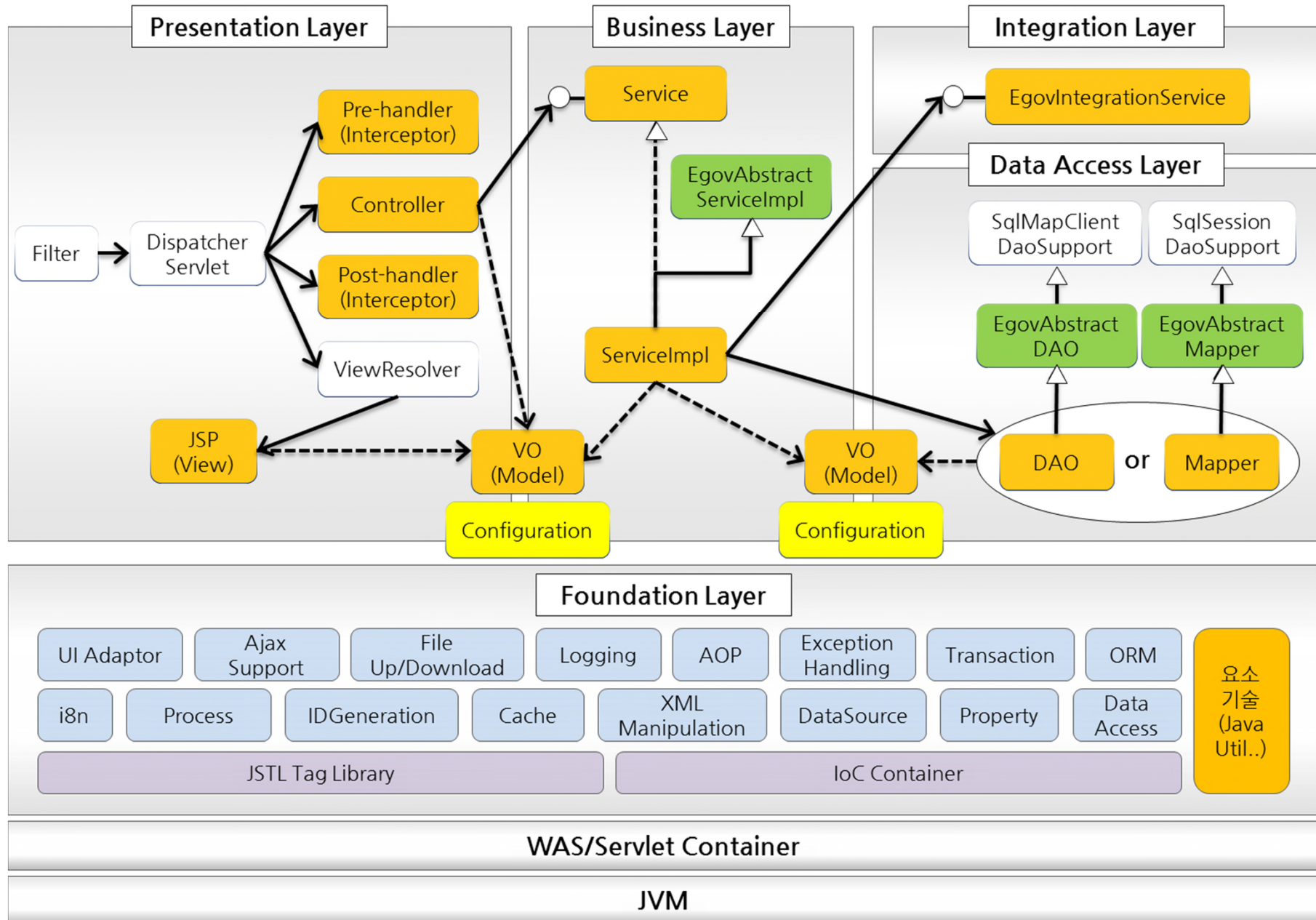


환경	설명
실행 환경	<ul style="list-style-type: none"> 업무 프로그램의 실행에 필요한 공통 모듈 및 업무 공통서비스를 제공함
개발 환경	<ul style="list-style-type: none"> 업무 프로그램에 대한 요구사항 수집, 분석/설계, 구현, 테스트, 배포 등의 개발 Life-Cycle 전반에 대한 지원 도구를 제공함
운영 환경	<ul style="list-style-type: none"> 업무 프로그램을 운영하기 위한 운영 도구를 제공함
관리 환경	<ul style="list-style-type: none"> 개발프레임워크의 지속적인 개선 및 유지보수를 효과적으로 수행하기 위한 관리 도구를 제공함

- ❑ 전자정부 개발프레임워크 실행환경은 응용 SW의 구성기반이 되며 응용 SW 실행 시 필요한 기본 기능을 제공하는 환경임
- ❑ 전자정부 개발프레임워크 실행환경은 8개 서비스 그룹으로 구성되며 39개 서비스를 제공함



1. 개요(3/3) – 개발 프레임워크 아키텍처 뷰



□ 기존 전자정부 프레임워크의 문제점

- 전자정부에 적용된 개발프레임워크는 Black Box 형태로 제공됨
- 사업자의 기술지원 없이는 응용 SW를 유지보수하기 어렵고 **사업자에 대한 의존성이 발생함**
- 개발프레임워크에 따라 개발표준 정의, 개발자수급, 교육시행 등 별도의 유지보수 체계를 가짐
- 개발프레임워크의 체계적인 관리절차의 미비로 동일 개발프레임워크라 하더라도 버전 관리 어려움

□ 전자정부 개발프레임워크의 표준화

- 사업자 고유 개발프레임워크에 대한 기술 종속성을 배제
- 프레임워크 표준화를 통한 응용 SW의 표준화와 품질, 재사용성을 향상
- 개발 프레임워크의 유지 보수 단일화를 통한 투자 효율성을 높임

❑ 8개 서비스 그룹, 39개 서비스로 구성

- 화면처리, 업무처리, 데이터처리, 연계통합, 공통기반, 배치처리, 모바일 화면처리, 모바일디바이스 API 실행환경
- MVC, [IoC 컨테이너](#), [AOP](#), [Data Access](#), Integration 등의 핵심 서비스 39개를 제공

❑ 전자정부 프로젝트에 최적화된 오픈 소스 소프트웨어 선정

- 39개 서비스 별 최적의 오픈 소스 소프트웨어 선정

❑ 경량화된 개발프레임워크로서 사실상 업계 표준에 가까운 [Spring](#) 프레임워크를 적용

- J2EE 표준을 준수하는 Spring 프레임워크
- 특정 업체의 WAS나 DBMS에 독립적인 환경을 제공

❑ [DI\(Dependency Injection\)](#) 방식의 의존 관계 처리

- Dependency Injection을 통해 개체나 컴포넌트 간의 의존성을 정의함으로써 변경용이성과 재사용성을 향상

❑ [AOP\(Aspect Oriented Programming\)](#) 지원

- 트랜잭션, 예외처리와 같은 공통 관심 대상을 분리하여 정의함

❑ MVC Model2 아키텍처 구조 제공 및 다양한 UI 클라이언트 연계 지원

- Spring MVC를 기반으로 하며, 다양한 UI 클라이언트 연계를 위한 인터페이스를 정의함

❑ 전자정부 개발프레임워크 표준 연계 인터페이스 정의

- 표준 연계 인터페이스를 정의하여 연계 솔루션에 대한 의존성을 배제하고 독립적인 어플리케이션 개발이 가능함

❑ 개발 생산성 향상

- 공통적으로 필요한 기능을 제공함으로써 개발 중복을 최소화하고 기반 구조를 정의함으로써 개발자가 비즈니스 업무에 집중할 수 있도록 함

❑ 전자정부 시스템의 재사용성 향상

- 전자정부 개발프레임워크에서 개발된 사업 컴포넌트를 공유함으로써 재사용성을 향상시킴

❑ 전자정부 상호운용성 향상

- 전자정부 개발프레임워크 사용 시스템간 연계 표준 인터페이스를 사용함으로써 상호 운용성이 향상됨

❑ 전자정부 응용 소프트웨어 표준화 효과

- 화면처리/업무처리/데이터처리의 표준화된 개발 기반을 제공함으로써 개발 코드의 표준화를 유도함

❑ 오픈 소스 활성화

- 오픈 소스에 기반한 표준 프레임워크를 정의함으로써 개발자들의 오픈 소스 사용을 활성화 함

❑ 중소 소프트웨어 사업자의 산업경쟁력 강화

- 전자정부 개발프레임워크를 공유하고 프레임워크 기술 인력을 증가시킴으로써 중소 소프트웨어 사업자의 경쟁력을 강화함

□ 화면처리

- 업무 프로그램과 사용자 간의 인터페이스를 담당하는 레이어, 사용자 화면 구성, 사용자 입력 정보 검증 등의 기능 제공

□ 업무처리

- 업무 프로그램의 업무 로직을 담당하는 레이어 업무 흐름 제어, 에러 처리 등의 기능 제공

□ 데이터처리

- DB에 대한 연결 및 영속성 처리, 트랜잭션 관리 제공

□ 연계통합

- 타 시스템과의 연동 기능을 제공

□ 공통기반

- 실행 환경 서비스에서 공통적으로 사용하는 기능 제공

□ 배치처리

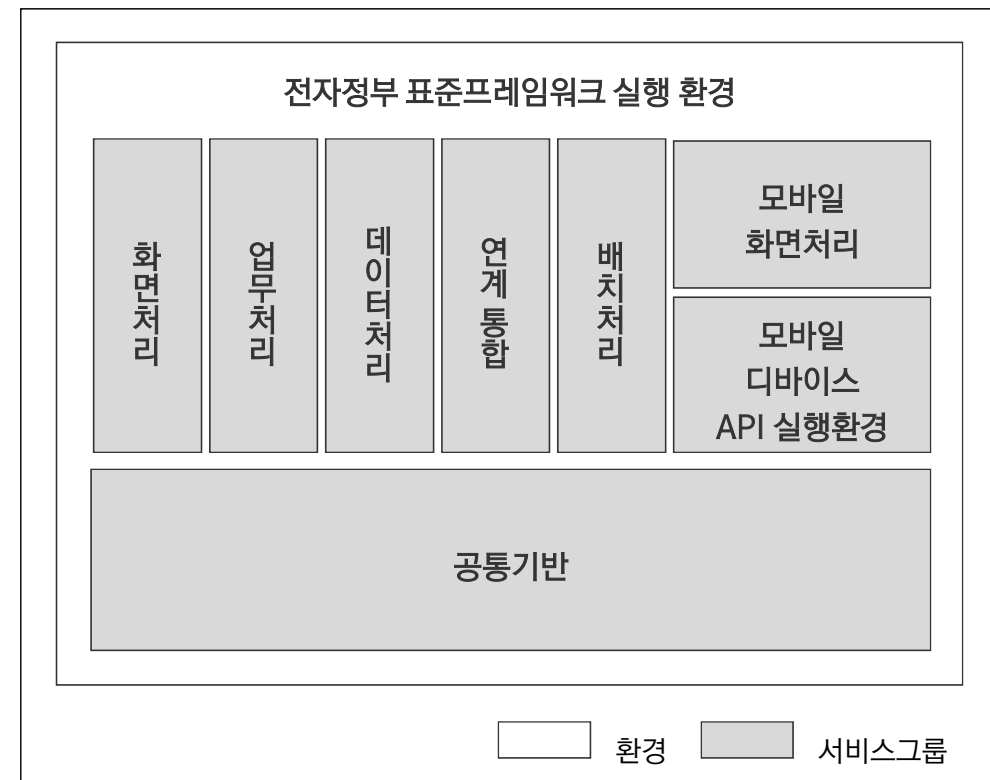
- 대용량 데이터처리 지원 작업수행 및 관리기능 제공

□ 모바일 화면처리

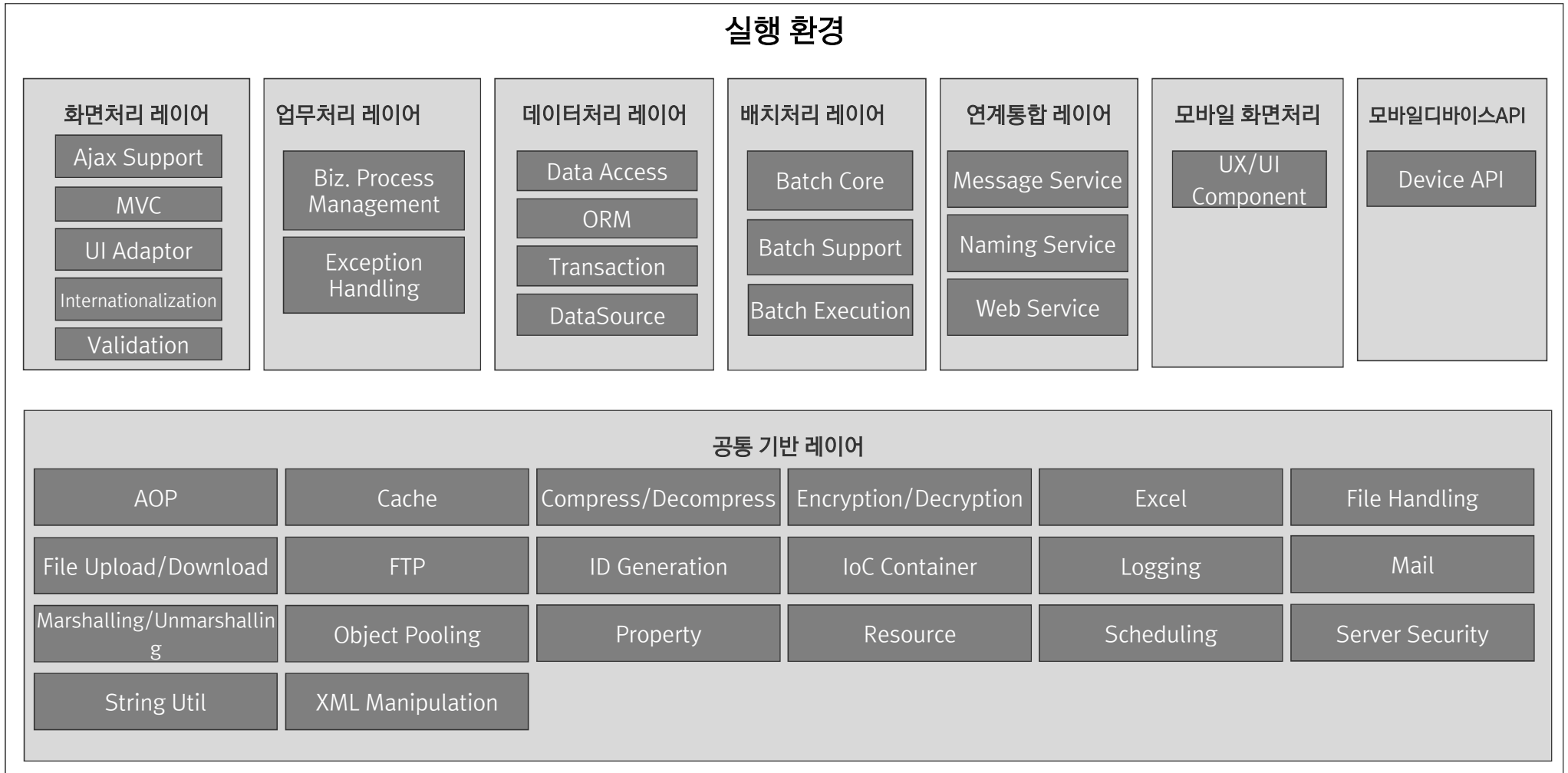
- 모바일 환경의 화면구성을 지원하는 사용자 경험 기반


□ 모바일 디바이스 API 실행환경

- 모바일 하이브리드 어플리케이션 개발을 위한 디바이스 API 기능 제공



- 전자정부 개발프레임워크 실행환경은 8개 서비스 그룹으로 구성되며 39개 서비스를 제공함



 공통기반 레이어 살펴보기

□ 화면처리 레이어

서비스 명	설명
MVC	MVC 디자인 패턴을 적용하여 사용자 화면을 개발할 수 있도록 MVC 기반 구조를 제공한다.
Internationalization	Internationalization은 다양한 지역과 언어 환경을 지원할 수 있는 서비스로, 서버 설정 및 클라이언트 브라우저 환경에 따라 자동화된 다국어 기능을 제공한다.
Ajax Support	Ajax는 대화식 웹 애플리케이션의 제작을 위해 HTML과 DOM, 자바 스크립트, XML, JSON 등과 같은 조합을 이용하는 웹 개발 기법으로 Ajax 기능 지원을 위한 JQuery 를 제공한다.
Validation	화면처리 레이어에서 사용자의 입력값 유효성을 client-side, server-side 환경 별로 검증 기능을 제공한다.
UI Adaptor	화면처리 레이어의 구현 방식에 따라 업무로직 레이어가 변경되는 것을 막기 위해서, 업무처리 Layer에서 사용할 데이터 타입을 정의하고, 화면 레이어에서 사용하는 in/out parameter를 해당 구현 방식에 맞게 변환해주는 기능을 제공한다

□ 업무처리 레이어

서비스 명	설명
Process Control	비즈니스 로직과 업무 흐름의 분리를 지원하며, XML 등의 외부 설정으로 업무흐름 구성을 제공하고, 미리 정의된 프로세스를 실행하는 기능을 제공한다.
Exception Handling	응용 프로그램의 실행 과정에서 발생하는 예외사항(Exception)을 처리하기 위해 표준화된 방법을 제공한다.

□ 연계통합 레이어

서비스 명	설명
Naming Service	원격에 있는 모듈 및 자원 등을 찾아주는 기능을 제공한다.
Web Service	연계 등을 위하여 업무 서비스를 웹 서비스의 형태로 어플리케이션 외부에 노출시켜 타 시스템이나 어플리케이션에서 서비스를 이용할 수 있도록 하는 기능을 제공한다.
Integration Service	전자정부 시스템과의 연계를 위한 공통 인터페이스를 제공하여, 다양한 솔루션 연계 방식에 대한 표준화된 처리 방식을 제공한다.

□ 데이터 처리 레이어

서비스 명	설명
Data Access	다양한 데이터베이스 솔루션 및 데이터베이스 접근 기술에 대한 추상화된 접근 방식을 제공하여 업무 로직과 데이터베이스 솔루션 및 접근 기술 간의 종속성을 배제하기 위한 기능을 제공한다.
Data Source	다양한 방식의 데이터베이스 연결을 제공하고, 이에 대한 추상화 계층을 제공함으로써, 업무 로직과 데이터베이스 연결 방식 간의 종속성을 배제하기 위한 기능을 제공한다.
ORM	객체 모델과 관계형 데이터베이스 간의 매핑 기능인 ORM(Object-Relational Mapping) 기능을 제공함으로써, SQL이 아닌 객체를 이용한 업무 로직의 작성이 가능하도록 지원한다.
Transaction	Database Transaction을 처리하기 위한 서비스로서, Transaction 처리에 대한 추상화된 방법을 제공하여 일관성 있는 프로그래밍 모델을 제공한다.

□ 공통기반 레이어(1/3)

서비스 명	설명
AOP	관점지향 프로그래밍(Aspect Oriented Programming: AOP) 사상을 구현하고 지원한다.
Cache	빈번히 사용되는 콘텐츠에 대해서 빠른 접근을 가능하게 하는 기능으로 잦은 접근을 통한 오버헤드나 시간을 단축시키는 역할을 한다.
Compress/Decompress	데이터를 압축 및 복원하는 기능을 제공한다. 데이터를 효율적으로 저장 및 전송하기 위해 원본 데이터를 압축하거나 압축된 데이터를 복원하여 원본 데이터를 구하는데 사용될 수 있다.
Encryption/Decryption	데이터에 대한 암호화 및 복호화 기능을 제공하며, 네트워크를 통한 데이터 송수신 시, 보안을 목적으로 사용될 수 있다.
Excel	엑셀 파일 포맷을 다룰 수 있는 자바 라이브러리를 제공한다.
File Handling	File 생성 및 접근, 변경을 위해 File에 Access할 수 있는 기능을 제공한다.
File Upload/Download	화면처리 서비스 그룹에서 사용되며, 파일을 업로드 및 다운로드 하기 위한 기능을 제공한다.
FTP	FTP(File Transfer Protocol) 프로토콜을 이용하여 데이터(파일)을 주고받기 위한 FTP 클라이언트 기능을 제공한다.

□ 공통기반 레이어(2/3)

서비스 명	설명
ID Generation	UUID(Universal Unique Identifier) 표준에 따라 시스템에서 사용하는 ID(Identifier)를 생성하는 기능을 제공한다.
IoC Container	프레임워크의 기본 기능인 IoC(Inversion of Control) 컨테이너 기능을 제공한다.
Logging	System.out.println 문을 사용한 오버헤드를 줄이고, 간편한 설정을 통해 로그를 저장하고 통제할 수 있는 기능을 제공한다.
Mail	SMTP 표준을 준수하며 이메일을 송신할 수 있도록 이메일 클라이언트 기능을 제공한다.
Marshalling/Unmarshalling	객체를 특정 데이터 형식으로 변환하고, 반대로 특정 데이터 형식으로 작성된 데이터를 객체로 변환하는 기능을 제공한다.
Object Pooling	Pool에 사용 가능한 객체가 있을 경우 객체를 할당 받거나, 없을 경우 Pool 크기에 따라 새로운 객체 생성 및 할당하는 기능을 제공한다.
Property	외부 파일이나 환경 정보를 구성하는 키와 값의 쌍을 내부적으로 저장하고 있으며, 어플리케이션이 특정 키에 대한 값에 접근할 수 있도록 기능을 제공한다.
Resource	국제화(Internationalization) 및 현지화(Localization)를 지원하기 위한 기능으로, 키 값을 이용하여 국가 및 언어에 해당하는 메시지를 읽어오는 기능을 제공한다.

□ 공통기반 레이어(3/3)

서비스 명	설명
Scheduling	어플리케이션 서버 내에서 주기적으로 발생하거나 반복적으로 발생하는 작업을 지원하는 기능으로서, 유닉스의 크론(Cron) 명령어와 유사한 기능을 제공한다.
Server Security	서버 함수 및 데이터 접근 시 보안 관리를 위해 사용자 인증 및 권한 관리 기능을 제공한다.
String Util	문자열 데이터를 다루기 위한 다양한 기능을 제공한다.
XML Manipulation	XML을 생성하고, 읽고, 쓰기 위한 기능을 제공한다.

□ 배치처리 레이어

서비스 명	설명
Batch Core	Job, Step 설정 및 실행, Reader/Writer 처리기능 등 , 배치 작업을 처리하는 기능을 제공한다.
Batch Execution	Job Repository, Job Runner, Job Launcher 실행방법 및 설정 기능을 제공한다.
Batch Support	병렬처리 기능 및 Listener를 사용한 Event Handling 기능 등을 제공한다.

□ 모바일 화면처리

서비스 명	설명
UX/UI Component	모바일 디바이스에 내의 모바일 브라우저 또는 웹 킷을 통한 화면 구성을 지원하는 UX/UI 컴포넌트를 제공한다.

□ 모바일 디바이스 API

서비스 명	설명
Device API	모바일 하이브리드 어플리케이션 구현 시 Javascript 를 이용하여, Device의 Native 기능에 접근이 가능도록 다양한 API를 제공한다.

□ 실행환경 오픈소스 소프트웨어 사용현황(1/4)

레이어	서비스 명	오픈소스 SW	확장 및 개발
화면처리	Ajax Support	Ajax Tags 1.5.7	
	Internationalization	Spring 4.3.22	
	MVC	Spring 4.3.22	Custom Tag 외 기능 확장
	Validation	Apache Commons Validator 1.4.0	
	UI Adaptor	선정하지 않음	UI Adaptor 연동 가이드
업무처리	Process Control	Spring Web Flow 2.4.0	3.0부터 개별적용
	Exception Handling	Spring 4.3.22	Exception 기능 확장
데이터 처리	Data Access	iBatis SQL Maps 2.3.4 MyBatis 3.4.6 Spring Data JPA 1.9.4	Spring-iBatis 기능 확장 MyBatis-Spring 기능확장 Spring-Data 기능확장
	DataSource	Spring 4.3.22	
	ORM	Hibernate 5.0.12 Final	
	Transaction	Spring 4.3.22	
연계통합	Naming Service Support	Spring 4.3.22	
	Integration Service	선정하지 않음	표준 인터페이스 처리 기능 개발
	Web Service Interface	Apache CXF 3.1.10	표준 인터페이스를 준수하도록 웹서비스를 확장

□ 실행환경 오픈소스 소프트웨어 사용현황(2/4)

레이어	서비스 명	오픈소스 SW	확장 및 개발
공통기반	AOP	Spring 4.3.22	
	Cache	EHCache 2.10.3 Spring 4.3.22	
	Compress/Decompress	Apache Commons Compress 1.8.1	
	Encryption/Decryption	Java simplified encryption (jasypt) 1.9.2	암호화 기능 확장
	Excel	Apache POI 3.10-FINAL jXLS 1.0.5	Excel 기능 확장 (xls, xlsx 지원)
	File Handling	Jakarta Commons VFS 2.0	File Access 기능 확장
	File Upload/Download	Commons FileUpload 2.0	
	FTP	Apache Commons Net 3.3	
	ID Generation	선정하지 않음	UUID 생성 기능 개발
	IoC Container	Spring 4.3.22	
	Logging	Log4j 2.10.2 SLF4J 1.7.25	SLF4J + Log4j 적용
	Mail	Apache Commons Email 1.3.2	
	Marshalling/Unmarshalling	Castor 4.3.22 Apache XML Beans 2.3.0	

□ 실행환경 오픈소스 소프트웨어 사용현황(3/4)

레이어	서비스 명	오픈소스 SW	확장 및 개발
공통기반 (계속)	Object Pooling	Apache Commons Pool 1.5.4	
	Property	Spring 4.3.22	Property 기능 확장
	Resource	Spring 4.3.22	
	Scheduling	Quartz 2.1.7	
	Server Security	Spring Security 4.2.11	인증, 권한 관리 기능 확장
	String Util	Jakarta Regexp 1.4	문자열 처리 기능 확장
	XML Manipulation	Apache Xerces 2.11.0 JDOM 2.0.5	XML 처리 기능 확장

□ 실행환경 오픈소스 소프트웨어 사용현황(4/4)

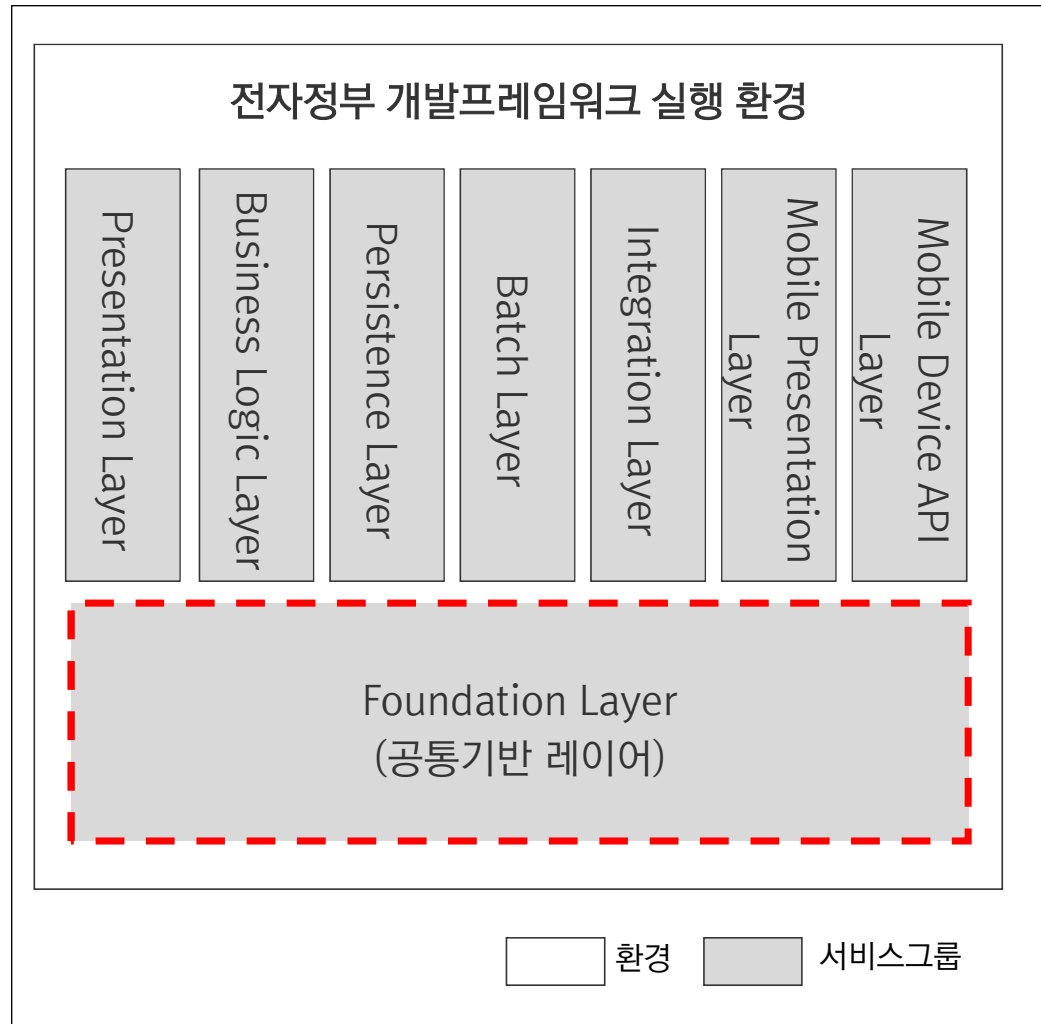
레이어	서비스 명	오픈소스 SW	확장 및 개발
배치처리	Batch Core	Spring Batch 3.0.6	Configuration의 기능 확장 및 데이터처리 성능향상
	Batch Execution		
	Batch Support		
모바일 화면처리	UX/UI Component	jQuery 1.12.4 jQuery Mobile 1.4.5	
모바일 디바이스 API	Device API	Cordova 8.1.2	



2. 공통기반 레이어

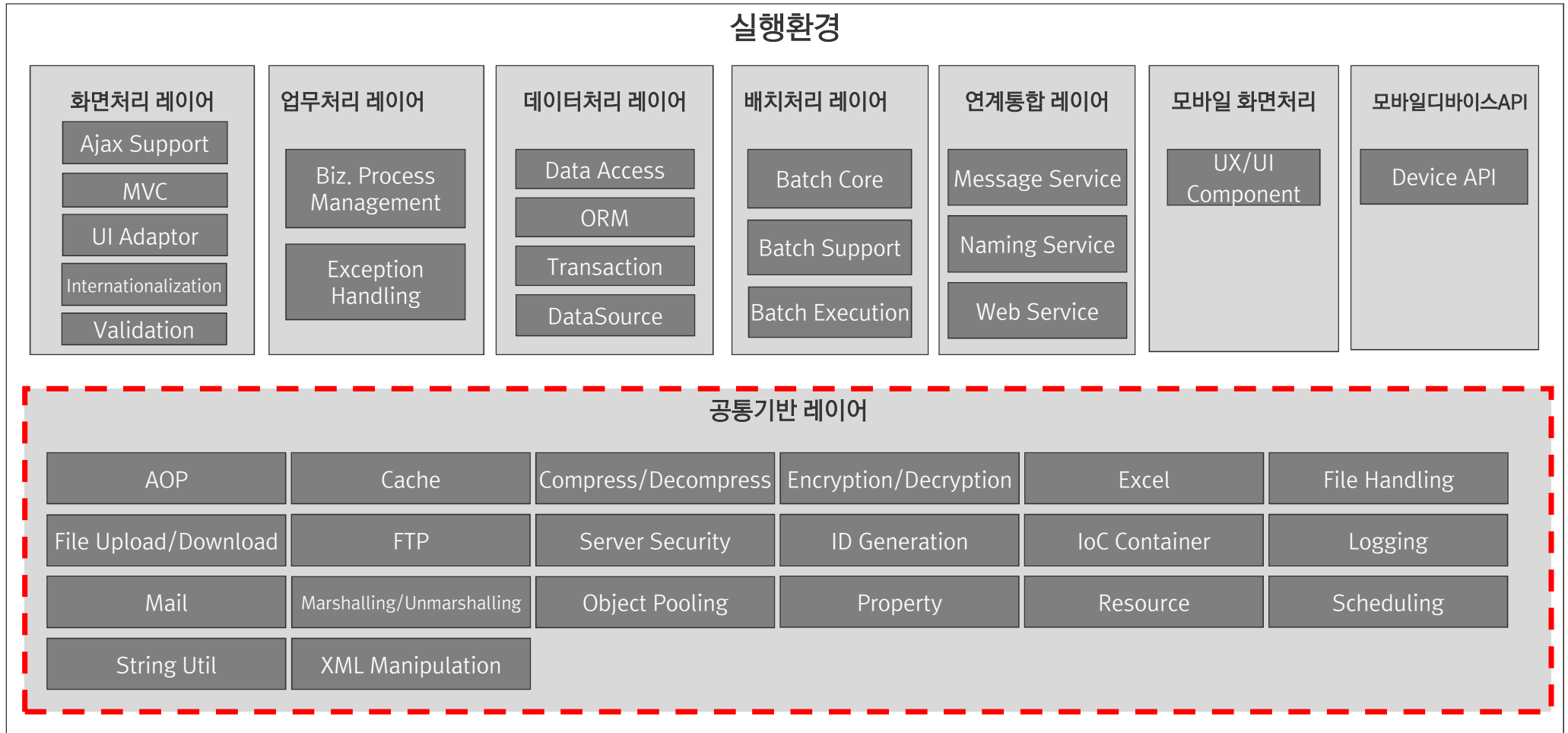
1. 개요
2. IOC
3. AOP
4. ID Generation
5. Logging

- 공통기반 레이어는 전자정부 개발프레임워크 실행환경 레이어 중의 하나로서 실행 환경의 각 Layer에서 공통적으로 사용하는 공통 기능을 제공함



서비스 그룹	설명
Presentation Layer	<ul style="list-style-type: none"> 업무 프로그램과 사용자 간의 Interface를 담당하는 Layer로서, 사용자 화면 구성, 사용자 입력 정보 검증 등의 기능을 제공함
Business Logic Layer	<ul style="list-style-type: none"> 업무 프로그램의 업무 로직을 담당하는 Layer로서, 업무 흐름 제어, 에러 처리 등의 기능을 제공함
Persistence Layer	<ul style="list-style-type: none"> 데이터베이스에 대한 연결 및 영속성 처리, 선언적인 트랜잭션 관리를 제공하는 Layer임
Batch Layer	<ul style="list-style-type: none"> 대용량 데이터 처리를 위한 기반 환경을 제공하는 Layer임
Integration Layer	<ul style="list-style-type: none"> 타 시스템과의 연동 기능을 제공하는 Layer임
Foundation Layer	<ul style="list-style-type: none"> 실행 환경의 각 Layer에서 공통적으로 사용하는 공통 기능을 제공함

- 공통기반 레이어는 **IoC, AOP, Security** 등 총 20개의 서비스를 제공함



실행환경
 서비스그룹
 서비스

□ 공통기반 레이어는 Spring, Quartz, EHCACHE 등 총 18종의 오픈소스 SW를 사용하고 있음

서비스	오픈소스 SW	오픈소스 버전
AOP	Spring	4.3.22
Cache	EHCACHE	2.10.3
Compress/Decompress	Apache Commons Compress	1.8.1
Encryption/Decryption	Java simplified encryption (JASYPT)	1.9.2
Excel	Apache POI	3.10-FINAL
File Handling	Jakarta Commons VFS	2.0
File Upload/Download	Apache Commons FileUpload	2.0
FTP	Apache Commons Net	3.3
IoC Container	Spring	4.3.22
Logging	Log4j	2.10.2

□ 공통기반 레이어는 Spring, Quartz, EHCACHE 등 총 18종의 오픈소스 SW를 사용하고 있음

서비스	오픈소스 SW	오픈소스 버전
Mail	Apache Commons Email	1.3.2
Marshalling/Unmarshalling	Castor Apache XMLBeans	4.3.22 2.3.0
Object Pooling	Apache Commons Pool	1.5.4
Property	Spring	4.3.22
Resource	Spring	4.3.22
Scheduling	Quartz	2.1.7
Server Security	Spring Security	4.2.11
String Util	Jakarta Regexp	1.4
XML Manipulation	Apache Xerces 2 JDOM	2.11.0 2.0.5

- ❑ Spring 프레임워크는 JavaEE 기반의 어플리케이션 개발을 쉽게 해주는 오픈소스 어플리케이션 프레임워크로, 간단한 자바 객체(POJO : Plain Old Java Object) 를 Spring의 경량(Lightweight) 컨테이너를 통해 생성 및 관리하는 빈(Been) 으로 처리해준다.



- 엔터프라이즈 어플리케이션을 쉽게 구성할 수 있도록 각종 빈(Been)의 생성 및 관리를 처리하는 경량(Lightweight) 컨테이너 제공
- Rod Johnson 에 의해 개발된 J2EE 어플리케이션 개발을 위한 오픈소스 어플리케이션 프레임워크

분류 및 성숙도 평가*	설명
라이선스	Apache 2.0
기능성 (Functionality)	✓✓✓✓ (중대형 규모의 기업의 기능적인 요구사항을 충족시킴)
커뮤니티 (Community)	*** (개발, 오류 보고, 수정 등의 활발한 커뮤니티 활동이 있음)
성숙도 (Maturity)	★★★★ (강력하며 높은 품질의 안정적이며 우수한 성능을 충족함)
적용성 (ER-Rating)	◆◆◆ (프레임워크가 성숙하여 기업 환경에 즉시 반영 가능함)
트렌드 (Trend)	↗ (평가 Criteria 전반적으로 발전하고 있으며, 중요도가 커지고 있음)

* Open Source Catalogue 2007, Optaros (Spring 2.0 기준)

□ EJB 명세와 현실의 괴리

- 원격 호출 기반의 EJB는 객체에 대해서 무거운(heavy weight)모델임
- 대부분의 개발자들은 스테이트리스 세션 빈과 비동기 방식이 필요한 경우에 한해서 메시지 드리븐 빈만을 사용함

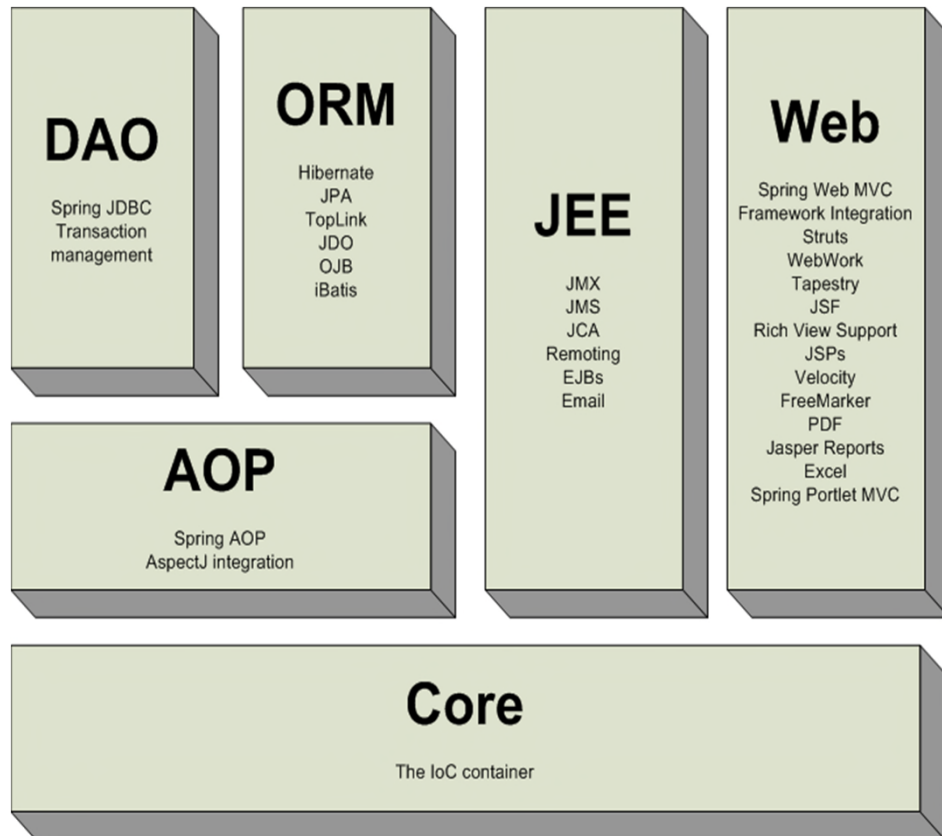
□ EJB 의 실패 부분

- 너무 복잡함
- 저장을 위한 엔티티 빈은 실패작임
- EJB의 이식성은 서블릿과 같은 다른 J2EE 기술보다 떨어짐
- 확장성을 보장한다는 EJB의 약속과 달리, 성능이 떨어지며 확장이 어려움

□ Spring Framework

- Spring이라는 이름의 기원은 전통적인 J2EE를 “겨울”에 빗대어 “겨울” 후의 “봄”으로 새로운 시작을 의미함
- Rod Johnson이 창시한 개발프레임워크
- EJB가 제공했던 대부분의 기능을 일반 POJO(Plain Old Java Object)를 사용하여 개발할 수 있도록 지원함
- 엔터프라이즈 어플리케이션 개발의 복잡성을 줄이기 위한 목적으로 개발됨

- Spring Framework는 어플리케이션을 구성하는 Bean 객체의 생명 주기를 관장하는 Core를 기반으로 DAO, ORM, AOP, JEE, Web으로 구성됨

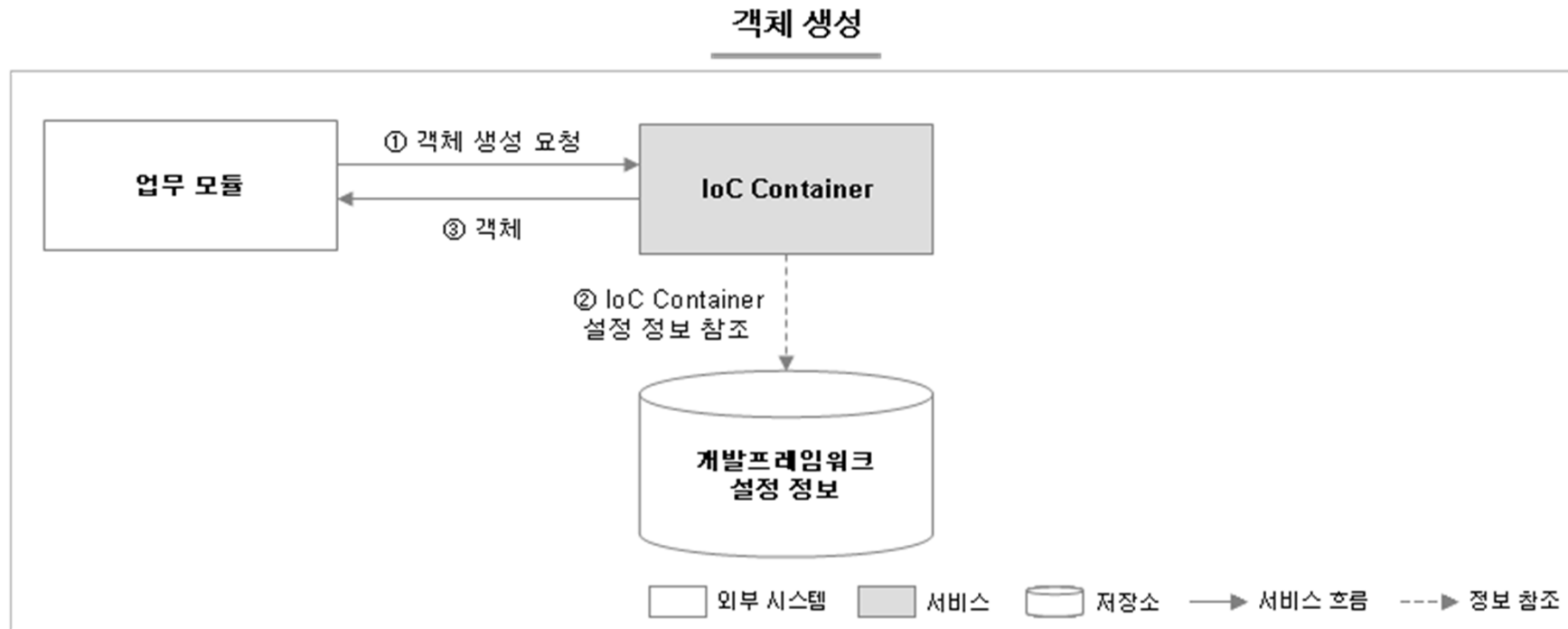


구분	설명
Core	<ul style="list-style-type: none"> Core 패키지는 프레임워크의 가장 기본적인 부분이고 IoC와 의존성 삽입(Dependency Injection-DI)기능을 제공한다.
DAO	<ul style="list-style-type: none"> DAO 패키지는 JDBC 코딩과 특정 데이터베이스 업체의 에러코드 파싱 등과 같은 작업의 필요성을 제거한 JDBC추상화 계층을 제공한다. 또한 프로그램적인 트랜잭션 관리 뿐 아니라 선언적인 트랜잭션 관리 기능을 제공한다.
ORM	<ul style="list-style-type: none"> ORM 패키지는 JAP, JDO, Hibernate, iBatis 등과 같은 객체-관계 매핑(Object-Relational Mapping)을 위한 통합 계층을 제공한다.
AOP	<ul style="list-style-type: none"> AOP 패키지는 AOP Alliance에서 정의한 Aspect-지향 프로그래밍 방식을 지원한다.
Web	<ul style="list-style-type: none"> Web 패키지는 멀티파트 파일업로드기능, 서블릿 리스너를 사용한 IoC컨테이너의 초기화, 웹-기반애플리케이션 컨텍스트 등과 같은 기본적인 웹-기반 통합 기능들을 제공한다.
JEE	<ul style="list-style-type: none"> Spring 환경에서 다양한 Java EE 기술을 사용할 수 있도록 지원한다.

□ 개요

- 프레임워크의 기본적인 기능인 IoC(Inversion of Control) Container 기능을 제공하는 서비스이다.
- 객체의 생성 시, 객체가 참조하고 있는 타 객체에 대한 의존성을 소스 코드 내부에서 하드 코딩하는 것이 아닌, 소스 코드 외부에서 설정하게 함으로써, 유연성 및 확장성을 향상시킨다.
- 주요 기능 : **Dependency Injection, Bean Lifecycle Management**
- 오픈 소스 : Spring Framework 4.0.의 IoC Container를 수정없이 사용함.
- **의존성 관리의 중요성**

□ 개요



- ① 업무 모듈은 IoC Container 서비스에 객체 생성을 요청한다.
- ② IoC Container는 표준 프레임워크 설정 정보에 객체 생성을 위한 종속성 정보 등과 같은 IoC Container 설정 정보를 참조한다.
- ③ IoC Container는 설정 정보에 따라 객체를 생성하여 업무 모듈에게 돌려준다.

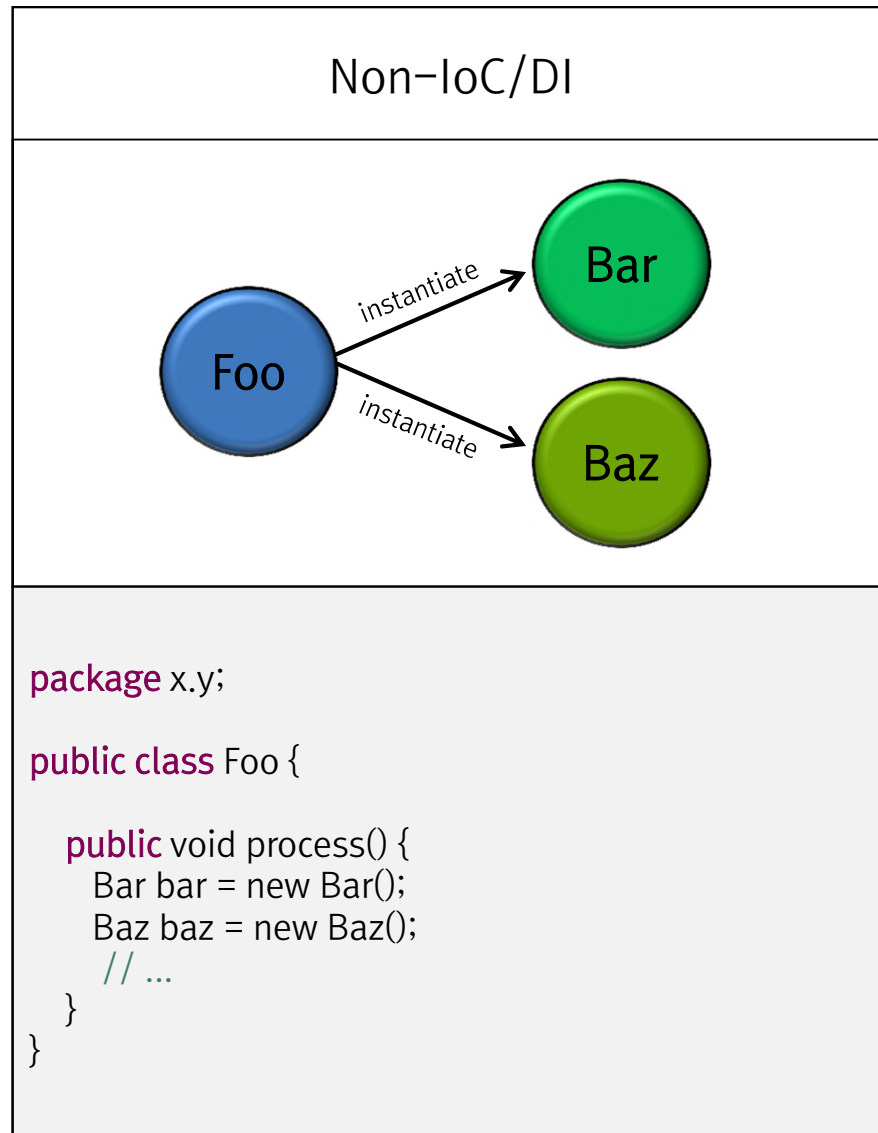
❑ IoC(Inversion of Control)이란?

- IoC는 Inversion of Control의 약자로 한글로 “제어의 역전” 또는 “역제어”라고 부른다. 어떤 모듈이 제어를 가진다는 것은 “어떤 모듈을 사용할 것인지”, “모듈의 함수는 언제 호출할 것인지” 등을 스스로 결정한다는 것을 의미한다. 이러한 제어가 역전되었다는 것은, 어떤 모듈이 사용할 모듈을 스스로 결정하는 것이 아니라 다른 모듈에게 선택권을 넘겨준다는 것을 의미한다.

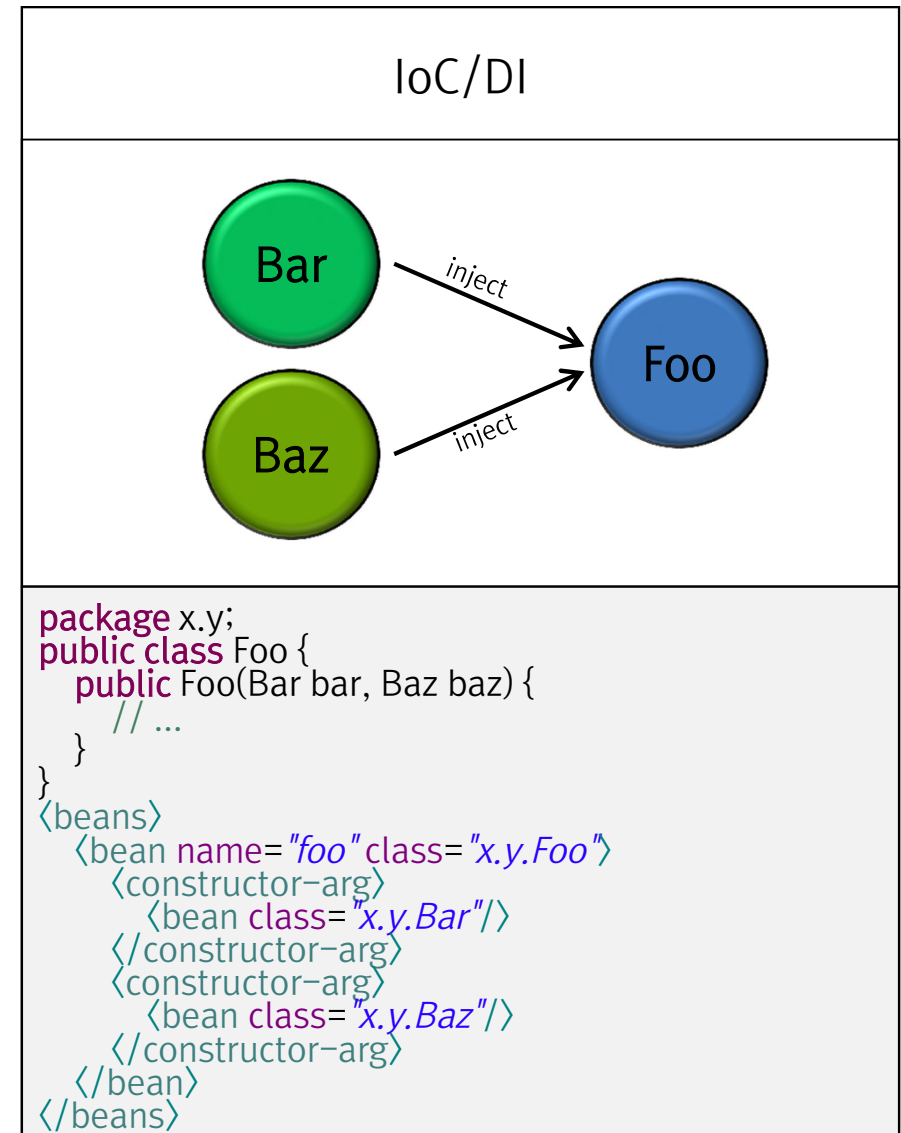
❑ DI(Dependency Injection)이란?

- Dependency Injection이란 모듈간의 의존성을 모듈의 외부(컨테이너)에서 주입시켜주는 기능으로 IoC의 한 종류이다.
- 런타임 시 사용하게 될 의존대상과의 관계를 Spring Framework 이 총체적으로 결정하고 그 결정된 의존특징을 런타임 시 부여한다.

□ Non-IoC/DI vs IoC/DI



VS



❑ Container

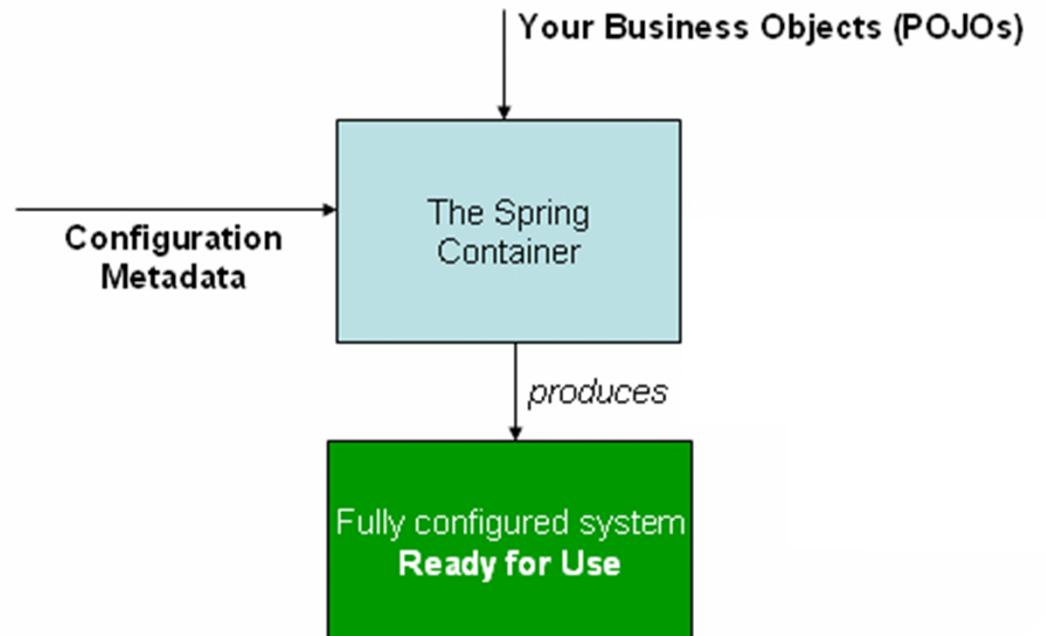
- Spring IoC Container는 객체를 생성하고, 객체 간의 의존성을 이어주는 역할을 한다.

❑ 설정 정보(Configuration Metadata)

- Spring IoC container가 “객체를 생성하고, 객체간의 의존성을 이어줄 수 있도록” 필요한 정보를 제공한다. 설정 정보는 기본적으로 XML 형태로 작성되며, 추가적으로 Java Annotation을 이용하여서도 설정이 가능하다.

❑ Bean

- Spring IoC Container에 의해 생성되고 관리되는 객체를 의미한다.



❑ BeanFactory

- BeanFactory 인터페이스는 Spring IoC Container의 기능을 정의하고 있는 기본 인터페이스이다.
- Bean 생성 및 의존성 주입, 생명주기 관리 등의 기능을 제공한다.

❑ ApplicationContext

- BeanFactory 인터페이스를 상속받는 ApplicationContext는 BeanFactory가 제공하는 기능 외에 Spring AOP, 메시지 리소스 처리(국제화에 사용됨), 이벤트 처리 등의 기능을 제공한다.
- 모든 ApplicationContext 구현체는 BeanFactory의 기능을 모두 제공하므로, 특별한 경우를 제외하고는 ApplicationContext를 사용하는 것이 바람직하다.
- Spring Framework는 다수의 ApplicationContext 구현체를 제공한다. 다음은 ClassPathXmlApplicationContext를 생성하는 예제이다.

```
ApplicationContext context = new ClassPathXmlApplicationContext(  
    new String[] {"services.xml", "daos.xml"});  
Foo foo = (Foo)context.getBean("foo");  
  
// an Application is also a BeanFactory (via inheritance)  
BeanFactory factory = context;
```

※ Spring Container = Bean Factory = ApplicationContext = DI Container = IoC Container

□ XML 설정 파일(1/2)

- XML 설정 파일은 <beans/> element를 root로 갖는다. 아래는 기본적인 XML 설정 파일의 모습이다.

```
<?xml version="1.0" encoding="UTF-8"?>
<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-4.0.xsd">

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

□ XML 설정 파일(2/2)

- XML 설정은 여러 개의 파일로 구성될 수 있으며, `<import/>` element를 사용하여 다른 XML 설정 파일을 import할 수 있다.

```
<beans>

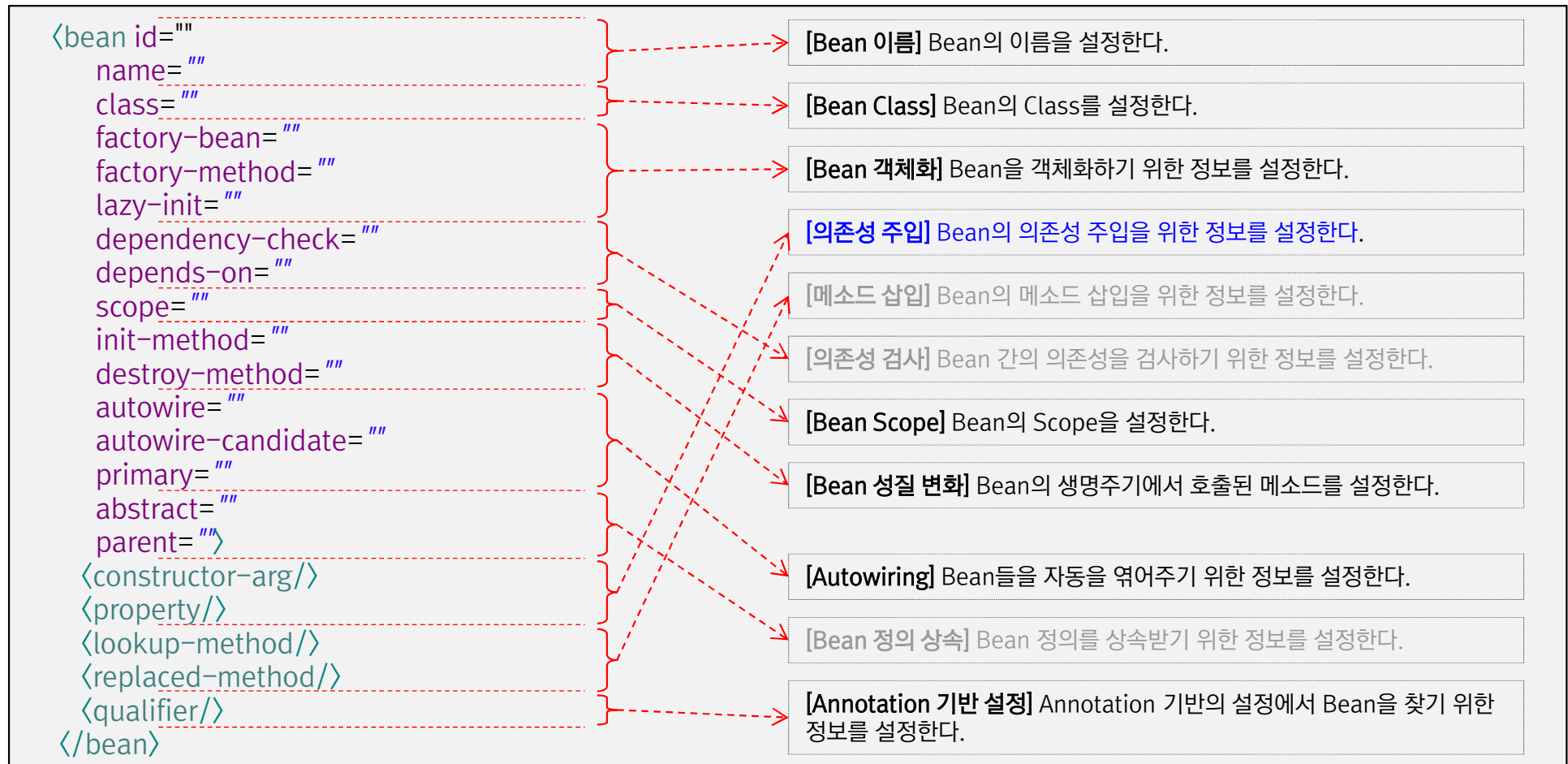
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>

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

</beans>
```

□ Bean 정의

- Bean 정의는 Bean을 객체화하고 의존성을 주입하는 등의 관리를 위한 정보를 담고 있다. XML 설정에서는 <bean/> element가 Bean 정의를 나타낸다. Bean 정의는 아래와 같은 속성을 가진다.



□ Bean 이름

- 모든 Bean은 하나의 id를 가지며, 하나 이상의 name을 가질 수 있다. id는 container 안에서 고유해야 한다.

```
<bean id="exampleBean" class="example.ExampleBean"/>  
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

- Bean은 <alias/> element를 이용하여 추가적인 name을 가질 수 있다.

```
<alias name="fromName" alias="toName"/>
```

□ Bean Class

- 모든 Bean은 객체화를 위한 Java class가 필요하다. (예외적으로 상속의 경우 class가 없어도 된다.)

```
<bean id="exampleBean" class="example.ExampleBean"/>  
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

□ Bean 객체화(1/2)

- 일반적으로 Bean 객체화는 Java 언어의 ‘new’ 연산자를 사용한다. 이 경우 별도의 설정은 필요없다.
- ‘new’ 연산자가 아닌 static factory 메소드를 사용하여 Bean을 객체화할 수 있다. 이 경우 Constructor Injection 방식의 의존성 주입 설정을 따른다.

```
<bean id="exampleBean"  
      class="examples.ExampleBean"  
      factory-method="createInstance"/>
```

- 자신의 static factory 메소드가 아닌 별도의 Factory 클래스의 static 메소드를 사용하여 Bean을 객체화할 수 있다. 이 경우 역시 Constructor Injection 방식의 의존성 주입 설정을 따른다.

```
<!-- the factory bean, which contains a method called createInstance() -->  
<bean id="serviceLocator" class="com.foo.DefaultServiceLocator">  
  <!-- inject any dependencies required by this locator bean -->  
</bean>  
  
<!-- the bean to be created via the factory bean -->  
<bean id="exampleBean"  
      factory-bean="serviceLocator"  
      factory-method="createInstance"/>
```

```
ExampleBean exampleBean = new ExampleBean();  
ExampleBean exampleBean = ExampleBean.createInstance();  
ExampleBean exampleBean = DefaultServiceLocator.createInstance();
```

□ Bean 객체화(2/2)

- `<bean/>` element의 'lazy-init' attribute를 사용하여 Bean 객체화 시기를 설정할 수 있다.
 - 일반적으로 Bean 객체화는 BeanFactory가 객체화되는 시점에 수행된다. 만약, 'lazy-init' attribute 값이 'true'인 경우, 설정된 Bean의 객체가 실제로 필요하다고 요청한 시점에 객체화가 수행된다.
 - 'lazy-init' attribute가 설정되어 있지 않으면 기본값을 사용한다. Spring Framework의 기본값은 'false'이다.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

- `<beans/>` element의 'default-lazy-init' attribute를 사용하여 XML 설정 파일 내의 모든 Bean 정의에 대한 lazy-init attribute의 기본값을 설정할 수 있다.

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

□ 의존성 주입(1/15)

- 의존성 주입에는 **Constructor Injection**과 **Setter Injection**의 두가지 방식이 있다.
- Constructor Injection(1/3)
 - Constructor Injection은 **argument**를 갖는 생성자를 사용하여 의존성을 주입하는 방식이다. **<constructor-arg/>** element를 사용한다. 생성자의 argument와 <constructor-arg/> element는 class가 같은 것끼리 매핑한다.

```
package x.y;  
  
public class Foo {  
  
    public Foo(Bar bar, Baz baz) {  
        // ...  
    }  
}
```

```
<beans>  
  <bean name="foo" class="x.y.Foo">  
    <constructor-arg>  
      <bean class="x.y.Bar"/>  
    </constructor-arg>  
    <constructor-arg>  
      <bean class="x.y.Baz"/>  
    </constructor-arg>  
  </bean>  
</beans>
```

□ 의존성 주입(2/15)

– Constructor Injection(2/3)

- 만약 생성자가 같은 class의 argument를 가졌거나 primitive type인 경우 argument와 <constructor-arg/> element간의 매핑이 불가능하다. 이 경우, **Type을 지정하거나 순서를 지정할 수 있다.**

```
package examples;

public class ExampleBean {

    // No. of years to the calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

□ 의존성 주입(3/15)

– Constructor Injection(3/3)

- Type 지정

```
<bean id="exampleBean" class="examples.ExampleBean">  
  <constructor-arg type="int" value="7500000"/>  
  <constructor-arg type="java.lang.String" value="42"/>  
</bean>
```

- 순서 지정

```
<bean id="exampleBean" class="examples.ExampleBean">  
  <constructor-arg index="0" value="7500000"/>  
  <constructor-arg index="1" value="42"/>  
</bean>
```

□ 의존성 주입(4/15)

– Setter Injection(1/2)

- Setter Injection은 **argument가 없는 기본 생성자를 사용하여 객체를 생성한 후, setter 메소드를 사용하여 의존성을 주입하는 방식으로, <property/> element를 사용한다.**
- Class에 attribute(또는 setter 메소드 명)과 <property/> element의 'name' attribute를 사용하여 매핑한다.

```
public class ExampleBean {  
  
    private AnotherBean beanOne;  
    private YetAnotherBean beanTwo;  
    private int i;  
  
    public void setBeanOne(AnotherBean beanOne) {  
        this.beanOne = beanOne;  
    }  
  
    public void setBeanTwo(YetAnotherBean beanTwo) {  
        this.beanTwo = beanTwo;  
    }  
  
    public void setIntegerProperty(int i) {  
        this.i = i;  
    }  
}
```

□ 의존성 주입(5/15)

– Setter Injection(2/2)

```
<bean id="exampleBean" class="examples.ExampleBean">
  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

- 매핑 규칙은 <property/> element의 'name' attribute의 첫문자를 알파벳 대문자로 변경하고 그 앞에 'set'을 붙인 setter 메소드를 호출한다.

```
<bean id="exampleBean"
  class="examples.ExampleBean">
  <property name="beanOne">
    <ref bean="anotherExampleBean"/>
  </property>
</bean>
```

```
public class ExampleBean {
  public void setBeanOne(
    AnotherBean beanOne)
  {
    this.beanOne = beanOne;
  }
}
```


□ 의존성 주입(6/15)

– 의존성 상세 설정(1/10)

- `<constructor-arg/>` element 과 `<property/>` element는 ‘명확한 값’, ‘다른 Bean에 대한 참조’, ‘Inner Bean’, ‘Collection’, ‘Null’ 등의 값을 가질 수 있다.
- 명확한 값

Java Primitive Type, String 등의 명확한 값을 나타낸다. 사람이 인식 가능한 문자열 형태를 값으로 갖는 `<value/>` element를 사용한다. Spring IoC container가 String 값을 해당하는 type으로 변환하여 주입해준다.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/mydb</value>
    </property>
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value>masterkaoli</value>
    </property>
</bean>
```

□ 의존성 주입(7/15)

– 의존성 상세 설정(2/10)

- 다른 Bean에 대한 참조

<ref/> element를 사용하여 다른 Bean 객체를 참조할 수 있다. 참조할 객체를 지정하는 방식은 ‘container’, ‘local’, ‘parent’ 등이 있다.

1. container : 가장 일반적인 방식으로 같은 container 또는 부모 container에서 객체를 찾는다.

```
<ref bean="someBean"/>
```

2. local : 같은 XML 설정 파일 내에 정의된 Bean 객체를 찾는다.

```
<ref local="someBean"/>
```

3. parent : 부모 XML 설정 파일 내에 정의된 Bean 객체를 찾는다.

```
<!-- 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="accountService" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <ref parent="accountService"/>
  </property>
</bean>
```

□ 의존성 주입(8/15)

– 의존성 상세 설정(3/10)

- Inner Bean

〈property/〉 또는 〈constructor-arg/〉 element 안에 있는 〈bean/〉 element를 inner bean이라고 한다. Inner bean의 'scope' flag 와 'id', 'name'은 무시된다. Inner bean의 scope은 항상 prototype이다. 따라서 inner bean을 다른 bean에 주입하는 것은 불가능하다.

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

□ 의존성 주입(9/15)

– 의존성 상세 설정(4/10)

• Collection(1/2)

Java Collection 타입인 List, Set, Map, Properties를 표현하기 위해 <list/>, <set/>, <map/>, <props/> element가 사용된다. map의 key와 value, set의 value의 값은 아래 element 중 하나가 될 수 있다.

bean ref idref list set map props value null
--

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

□ 의존성 주입(10/15)

– 의존성 상세 설정(5/10)

• Collection(2/2)

```
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
    <entry>
      <key><value>an entry</value></key>
      <value>just some string</value>
    </entry>
    <entry>
      <key><value>a ref</value></key>
      <ref bean="myDataSource" />
    </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>
</bean>
```

□ 의존성 주입(11/15)

– 의존성 상세 설정(6/10)

- Null

Java의 null 값을 사용하기 위해서 <null/> element를 사용한다. Spring IoC container는 value 값이 설정되어 있지 않은 경우 빈문자열(“”)로 인식한다.

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

위 ExampleBean의 email 값은 “”이다. 아래는 email의 값이 null인 예제이다.

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

□ 의존성 주입(12/15)

– 의존성 상세 설정(7/10)

- 간편한 표기 1

`<property/>`, `<constructor-arg/>`, `<entry/>` element의 `<value/>` element는 'value' attribute로 대체될 수 있다.

```
<property name="myProperty">  
  <value>hello</value>  
</property>
```

=

```
<property name="myProperty" value="hello"/>
```

```
<constructor-arg>  
  <value>hello</value>  
</constructor-arg>
```

=

```
<constructor-arg value="hello"/>
```

```
<entry key="myKey">  
  <value>hello</value>  
</entry>
```

=

```
<entry key="myKey" value="hello"/>
```

□ 의존성 주입(13/15)

– 의존성 상세 설정(8/10)

• 간편한 표기 2

`<property/>`, `<constructor-arg/>` element의 `<ref/>` element는 'ref' attribute로 대체될 수 있다.

```
<property name="myProperty">
  <ref bean="myBean">
</property>
```

=

```
<property name="myProperty" ref="myBean"/>
```

```
<constructor-arg>
  <ref bean="myBean">
</constructor-arg>
```

=

```
<constructor-arg ref="myBean"/>
```

• 간편한 표기 3

`<entry/>` element의 'key', 'ref' element 는 'key-ref', 'value-ref' attribute로 대체될 수 있다.

```
<entry>
  <key>
    <ref bean="myKeyBean" />
  </key>
  <ref bean="myValueBean" />
</entry>
```

=

```
<entry key-ref="myKeyBean"
  value-ref="myValueBean"/>
```


□ 의존성 주입(14/15)

– 의존성 상세 설정(9/10)

- p-namespace(1/2)

〈property/〉 element 대신 ‘p-namespace’를 사용하여 XML 설정을 작성할 수 있다. 아래 classic bean과 p-namespace bean은 동일한 Bean 설정이다.

```
<beans ...>

  <bean name="classic" class="com.example.ExampleBean">
    <property name="email" value="foo@bar.com"/>
  </bean>

</beans>
```

=

```
<beans ...>

  <bean name="classic" class="com.example.ExampleBean"
    p:email="foo@bar.com"/>

</beans>
```

□ 의존성 주입(15/15)

– 의존성 상세 설정(10/10)

- p-namespace(2/2)

Attribute 이름 끝에 '-ref'를 붙이면 참조로 인식한다.

```
<beans ...>

  <bean name="john-classic" class="com.example.Person">
    <property name="name" value="John Doe"/>
    <property name="spouse" ref="jane"/>
  </bean>

  <bean name="jane" class="com.example.Person">
    <property name="name" value="Jane Doe"/>
  </bean>

</beans>
```

=

```
<beans ...>

  <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

  <bean name="jane" class="com.example.Person">
    <property name="name" value="Jane Doe"/>
  </bean>

</beans>
```

□ Autowiring

- Spring IoC container는 서로 관련된 Bean 객체를 자동으로 엮어줄 수 있다.
자동엮기(autowiring)는 각각의 bean 단위로 설정되며, 자동엮기 기능을 사용하면 `<property/>`나 `<constructor-arg/>`를 지정할 필요가 없어지므로, 타이핑일 줄일 수 있다.
- 자동엮기에는 5가지 모드가 있으며, XML 기반 설정에서는 `<bean/>` element의 'autowire' attribute를 사용하여 설정할 수 있다.

Scope	설명
no	자동엮기를 사용하지 않는다. Bean에 대한 참조는 <code><ref/></code> element를 사용하여 지정해야만 한다. 이 모드가 기본(default)이다.
byName	Property의 이름으로 자동엮기를 수행한다. Property의 이름과 같은 이름을 가진 bean을 찾아서 엮어준다.
byType	Property의 타입으로 자동엮기를 수행한다. Property의 타입과 같은 타입을 가진 bean을 찾아서 엮어준다. 만약 같은 타입을 가진 bean이 container에 둘 이상 존재할 경우 exception이 발생한다. 만약 같은 타입을 가진 bean이 존재하지 않는 경우, 아무 일도 발생하지 않는다. 즉, property에는 설정되지 않는다.
constructor	<i>byType</i> 과 유사하지만, 생성자 argument에만 적용된다. 만약 같은 타입의 bean이 존재하지 않거나 둘 이상 존재할 경우, exception이 발생한다.
autodetect	Bean class의 성질에 따라 <i>constructor</i> 와 <i>byType</i> 모드 중 하나를 선택한다. 만약 default 생성자가 존재하면, <i>byType</i> 모드가 적용된다. *Spring 3.0.4 버전부터 deprecated

- `<bean/>` element의 'autowire-candidate' attribute 값을 'false'로 설정함으로써, 대상 bean이 다른 bean과 자동으로 엮이는 것을 방지한다.

□ Bean Scope(1/4)

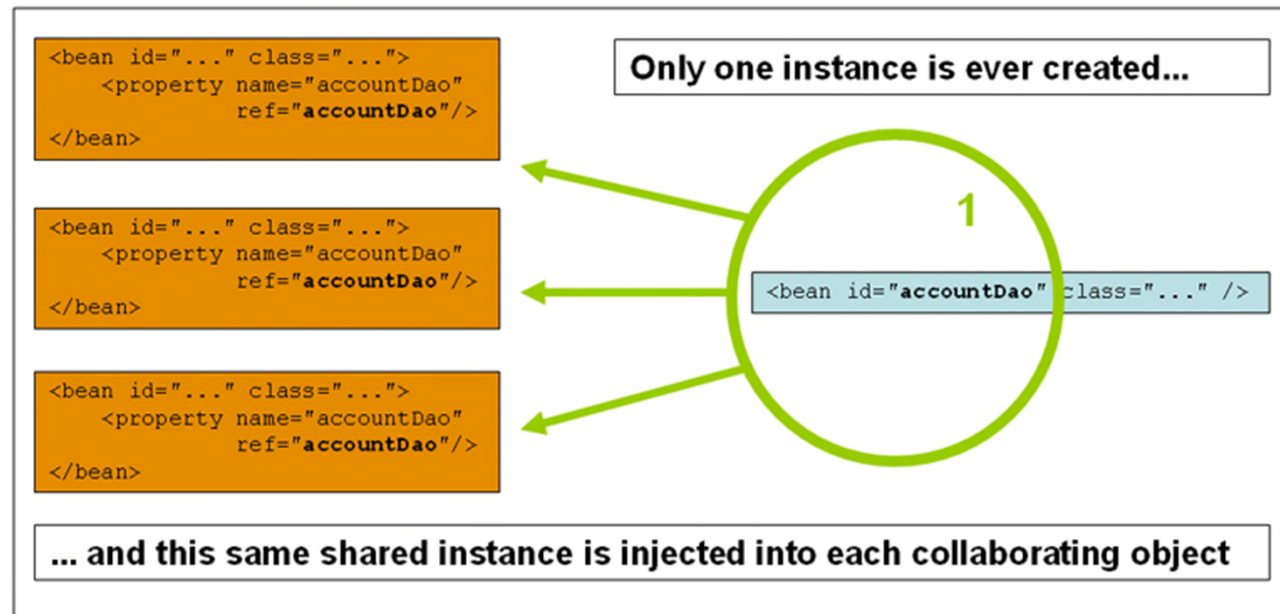
- Bean Scope은 객체가 유효한 범위로 아래 5가지의 scope이 있다.
- Spring Bean의 기본 Scope는 **singleton** 이다.

Scope	설 명
singleton	하나의 Bean 정의에 대해서 Spring IoC Container 내에 단 하나의 객체만 존재한다.
prototype	하나의 Bean 정의에 대해서 다수의 객체가 존재할 수 있다.
request	하나의 Bean 정의에 대해서 하나의 HTTP request의 생명주기 안에 단 하나의 객체만 존재한다; 즉, 각각의 HTTP request는 자신만의 객체를 가진다. Web-aware Spring ApplicationContext 안에서만 유효하다.
session	하나의 Bean 정의에 대해서 하나의 HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. Web-aware Spring ApplicationContext 안에서만 유효하다.
global session	하나의 Bean 정의에 대해서 하나의 global HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. 일반적으로 portlet context 안에서 유효하다. Web-aware Spring ApplicationContext 안에서만 유효하다.

□ Bean Scope(2/4)

– Singleton Scope

- Bean이 singleton인 경우, 단 하나의 객체만 공유된다.



- Spring의 기본 scope은 'singleton'이다. 설정하는 방법은 아래와 같다.

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>
```

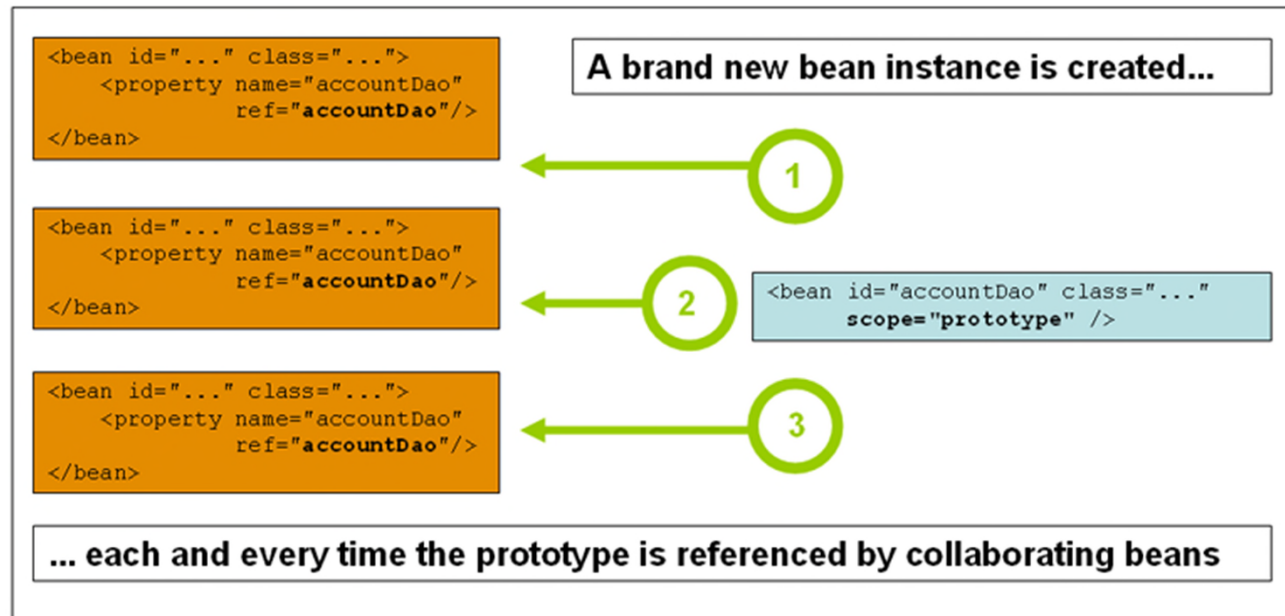
```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

```
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>
```

□ Bean Scope(3/4)

– Prototype Scope

- Singleton이 아닌 prototype scope의 형태로 정의된 **bean은 필요한 매 순간 새로운 bean 객체가 생성된다.**



- 설정하는 방법은 아래와 같다.

```

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>

```

□ Bean Scope(4/4)

– 기타 Scope

- request, session, global session은 Web 기반 어플리케이션에서만 유효한 scope이다.

- Request Scope

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

위 정의에 따라, Spring container는 모든 HTTP request에 대해서 'loginAction' bean 정의에 대한 LoginAction 객체를 생성할 것이다. 즉, 'loginAction' bean은 HTTP request 수준에 한정된다(scoped). 요청에 대한 처리가 완료되었을 때, 한정된(scoped) bean도 폐기된다.

- Session Scope

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

위 정의에 따라, Spring container는 하나의 HTTP Session 일생동안 'userPreferences' bean 정의에 대한 UserPreferences 객체를 생성할 것이다. 즉, 'userPreferences' bean은 HTTP Session 수준에 한정된다(scoped). HTTP Session이 폐기될 때, 한정된(scoped) bean도 폐기된다.

- Global Session Scope

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

global session scope은 HTTP Session scope과 비슷하지만 단지 portlet-based web 어플리케이션에서만 사용할 수 있다. Portlet 명세(specifications)는 global Session을 하나의 portlet web 어플리케이션을 구성하는 여러 portlet들 모두가 공유하는 것으로 정의하고 있다. global session scope으로 설정된 bean은 global portlet Session의 일생에 한정된다.

□ Bean 성질 변화(1/3)

– Lifecycle Callback

- Spring IoC Container는 **Bean의 각 생명주기에 호출되도록 설정된 메소드를 호출해준다.**

- Initialization callback

org.springframework.beans.factory.InitializingBean interface를 구현하면 bean에 필요한 모든 property를 설정한 후, 초기화 작업을 수행한다. InitializingBean interface는 다음 메소드를 명시하고 있다.

```
void afterPropertiesSet() throws Exception;
```

일반적으로, InitializingBean interface의 사용을 권장하지 않는다. 왜냐하면 code가 불필요하게 Spring과 결합되기(couple) 때문이다. 대안으로, bean 정의는 초기화 메소드를 지정할 수 있다. XML 기반 설정의 경우, 'init-method' attribute를 사용한다.

```
public class ExampleBean {  
  
    public void init() {  
        // do some initialization work  
    }  
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```


□ Bean 성질 변화(2/3)

– Lifecycle Callback

- Destruction callback

org.springframework.beans.factory.DisposableBean interface를 구현하면, container가 파괴될 때 bean이 callback를 받을 수 있다. DisposableBean interface는 다음 메소드를 명시하고 있다.

```
void destroy() throws Exception;
```

일반적으로, DisposableBean interface의 사용을 권장하지 않는다. 왜냐하면 code가 불필요하게 Spring과 결합되기(couple) 때문이다. 대안으로, bean 정의는 초기화 메소드를 지정할 수 있다. XML 기반 설정의 경우, 'destroy-method' attribute를 사용한다.

```
public class ExampleBean {  
    public void cleanup() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

□ Bean 성질 변화(3/3)

– Lifecycle Callback

- 기본 Initialization & Destruction callback

Spring IoC container 레벨에서 기본 Initialization & Destruction callback 메소드를 지정할 수 있다. <beans/> element의 'default-init-method', 'default-destroy-method' attribute를 사용한다.

```
<beans default-init-method="init" default-destroy-method="destroy">

  <bean id="blogService" class="com.foo.DefaultBlogService">
    <property name="blogDao" ref="blogDao" />
  </bean>

</beans>
```

□ Bean Profile (표준프레임워크 3.0부터 추가)

– Bean의 선택적 사용

- Bean profile

Profile을 이용하여 bean을 중복으로 선언하고 실제 사용시 선택적으로 사용하도록 설정할 수 있다. Bean Profile설정은 <beans profile/> element를 이용한다.

```
<beans profile="dev">
  <jdbc:embedded-database id="dataSource"> ... 생략
</jdbc:embedded-database>
</beans>

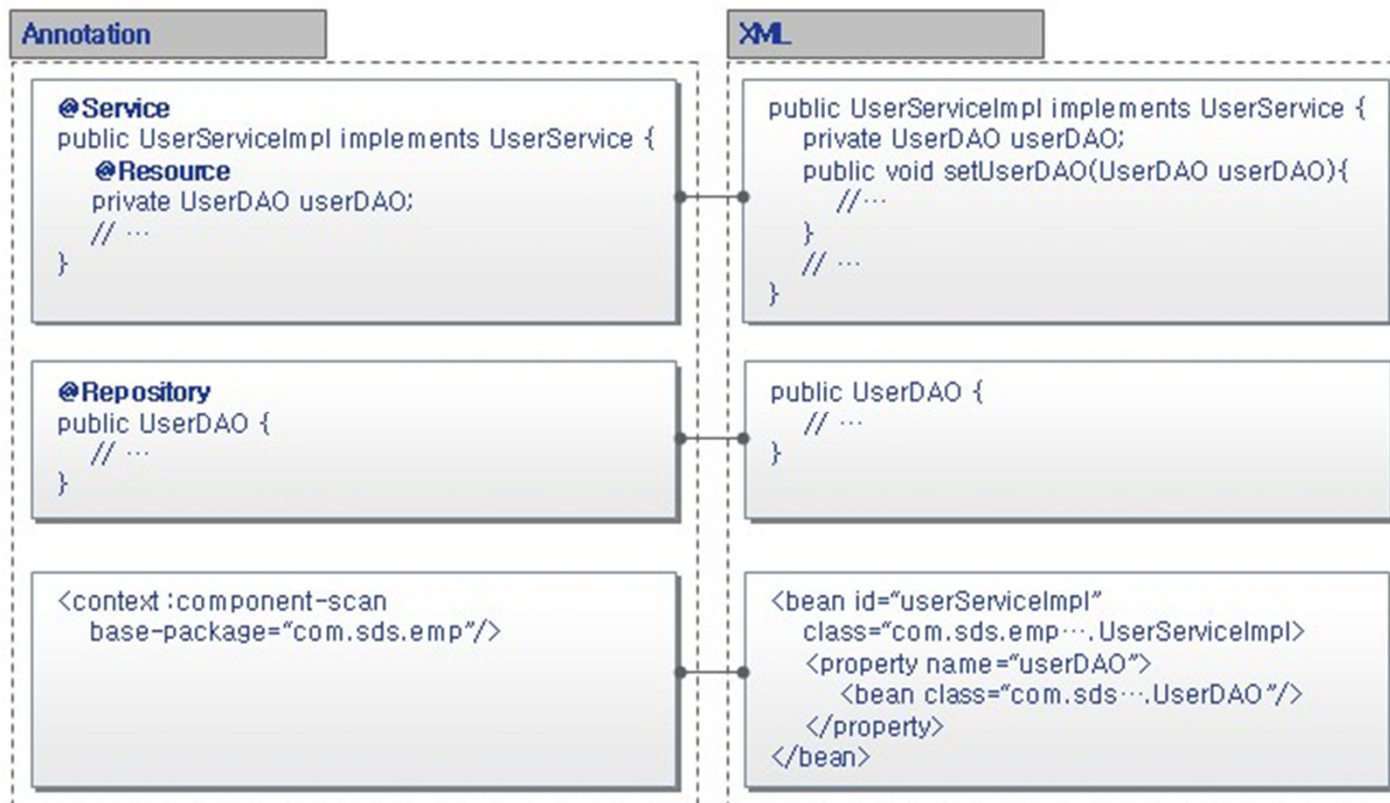
<beans profile="production">
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"> ... 생략
</bean>
</beans>
```

- Profile별 bean설정을 적용하기 위해서는 web.xml의 context-param (spring.profiles.active의 param-value)을 설정해주어야 한다.

```
<context-param>
  <param-name>spring.profiles.active</param-name>
  <param-value>production</param-value>
</context-param>
```

□ Annotaion

- XML 설정 파일을 사용하는 대신 자바 어노테이션을 사용할 수 있음 (자바 5 이상)
- annotation의 사용으로 설정파일을 간결화하고, View 페이지와 객체 또는 메소드의 맵핑을 명확하게 할 수 있다



□ Annotation 기반 설정(1/13)

- Spring은 Java Annotation을 사용하여 Bean 정의를 설정할 수 있다.

@Required, @Autowired, @Qualifier, @Resource, @PostConstruct, @PreDestroy 기능을 사용하기 위해서는 다음 namespace와 element를 추가해야 한다.

```
<?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/>

</beans>
```

□ Annotation 기반 설정(2/13)

– @Required

- @Required annotation은 setter 메소드에 적용된다. @Required annotation이 설정된 property는 <property/>, <constructor-arg/> element를 통해서 명시적으로 값이 설정되거나, autowiring에 의해서 값이 설정되어야 한다.

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Required  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

□ Annotation 기반 설정(3/13)

– @Autowired(1/3)

- @Autowired annotation은 자동으로 역을 property를 지정하기 위해 사용한다. setter 메소드, 일반적인 메소드, 생성자, field 등에 적용된다.
- Setter 메소드

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Autowired  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

□ Annotation 기반 설정(4/13)

– @Autowired(2/3)

- 일반적인 메소드

```
public class MovieRecommender {  
  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(MovieCatalog movieCatalog, CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```


□ Annotation 기반 설정(5/13)

– @Autowired(3/3)

- 생성자 및 field

```
public class MovieRecommender {  
  
    @Autowired  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

□ Annotation 기반 설정(6/13)

– @Qualifier(1/3)

- @Autowired annotation만을 사용하는 경우, 같은 Type의 Bean이 둘 이상 존재할 때 문제가 발생한다. 이를 방지하기 위해서 @Qualifier annotation을 사용하여 찾을 Bean의 대상 집합을 좁힐 수 있다. @Qualifier annotation은 field 뿐 아니라 생성자 또는 메소드의 parameter에도 사용할 수 있다.
- Field

```
public class MovieRecommender {  
  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
  
    // ...  
}
```

□ Annotation 기반 설정(7/13)

– @Qualifier(2/3)

- 메소드 Parameter

```
public class MovieRecommender {  
  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,  
                        CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

□ Annotation 기반 설정(8/13)

– @Qualifier(3/3)

- @Qualifier annotation의 값으로 사용되는 qualifier는 <bean/> element의 <qualifier/> element로 설정한다.

```
<?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="example.SimpleMovieCatalog">
    <qualifier value="main"/>
    <!-- inject any dependencies required by this bean -->
  </bean>

  <bean class="example.SimpleMovieCatalog">
    <qualifier value="action"/>
    <!-- inject any dependencies required by this bean -->
  </bean>

  <bean id="movieRecommender" class="example.MovieRecommender"/>
</beans>
```

□ Annotation 기반 설정(9/13)

– @Resource

- @Resource annotation의 name 값으로 대상 bean을 찾을 수 있다. @Resource annotation은 field 또는 메소드에 사용할 수 있다.

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource(name="myMovieFinder")  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

- @Resource annotation에 name 값이 없을 경우, field 명 또는 메소드 명을 이용하여 대상 bean을 찾는다.

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

□ Annotation 기반 설정(10/13)

– @PostConstruct & @PreDestroy

- @PostConstruct와 @PreDestroy는 각각 Instantiation callback, Destruction callback 메소드를 지정하기 위해 사용한다.

```
public class CachingMovieLister {  
  
    @PostConstruct  
    public void populateMovieCache() {  
        // populates the movie cache upon initialization...  
    }  
  
    @PreDestroy  
    public void clearMovieCache() {  
        // clears the movie cache upon destruction...  
    }  
}
```

□ Annotation 기반 설정(11/13)

– Auto-detecting components(1/3)

- Spring은 @Repository, @Service, @Controller annotation을 사용하여, 각각 Persistence, Service, Presentation 레이어의 컴포넌트로 지정하여 특별한 관리 기능을 제공하고 있다. @Repository, @Service, @Controller는 @Component annotation을 상속받고 있다. Spring IoC Container는 @Component annotation(또는 자손)으로 지정된 class를 XML Bean 정의 없이 자동으로 찾을 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:component-scan base-package="org.example"/>
</beans>
```

기본 package를 지정할 수 있다.

- <context:component-scan/>을 쓰는 경우에는 <context:annotation-config/>를 별도로 쓰지 않아도 된다.

□ Annotation 기반 설정(12/13)

– Auto-detecting components(2/3)

- 이름 설정

@Component, @Repository, @Service, @Controller annotation의 name 값으로 bean의 이름을 지정할 수 있다. 아래 예제의 Bean 이름은 “myMovieLister”이다.

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

만약 name 값을 지정하지 않으면, class 이름의 첫문자를 소문자로 변환하여 Bean 이름을 자동으로 생성한다. 아래 예제의 Bean 이름은 “movieFinderImpl”이다.

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```


□ Annotation 기반 설정(13/13)

– Auto-detecting components(3/3)

- Scope 설정

@Scope annotation을 사용하여, 자동으로 찾은 Bean의 scope를 설정할 수 있다.

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

- Qualifier 설정

@Qualifier annotation을 사용하여, 자동으로 찾은 Bean의 qualifier를 설정할 수 있다.

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

□ ApplicationContext for web application(1/2)

- Spring은 Web Application에서 ApplicationContext를 쉽게 사용할 수 있도록 각종 class들을 제공하고 있다.
- Servlet 2.4 이상
 - Servlet 2.4 specification부터 Listener를 사용할 수 있다. Listener를 사용하여 ApplicationContext를 설정하기 위해서 web.xml 파일에 다음 설정을 추가한다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

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

contextParam의 'contextConfigLocation' 값은 Spring XML 설정 파일의 위치를 나타낸다.

□ ApplicationContext for web application(2/2)

– Servlet 2.3 이하

- Servlet 2.3이하에서는 Listener를 사용할 수 없으므로, Servlet를 사용한다. web.xml 파일에 다음 설정을 추가한다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

<servlet>
  <servlet-name>context</servlet-name>
  <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```


□ 주요 개념

– Join Point

- 횡단 관심(Crosscutting Concerns) 모듈이 삽입되어 동작할 수 있는 실행 가능한 특정 위치를 말함
- 메소드 호출, 메소드 실행 자체, 클래스 초기화, 객체 생성 시점 등

– Pointcut

- Pointcut은 어떤 클래스의 어느 JoinPoint를 사용할 것인지를 결정하는 선택 기능을 말함
- 가장 일반적인 Pointcut은 ‘특정 클래스에 있는 모든 메소드 호출’로 구성된다.

– 애스펙트(Aspect)

- 어플리케이션이 가지고 있어야 할 로직과 그것을 실행해야 하는 지점을 정의한 것
- Advice와 Pointcut의 조합

– Advice

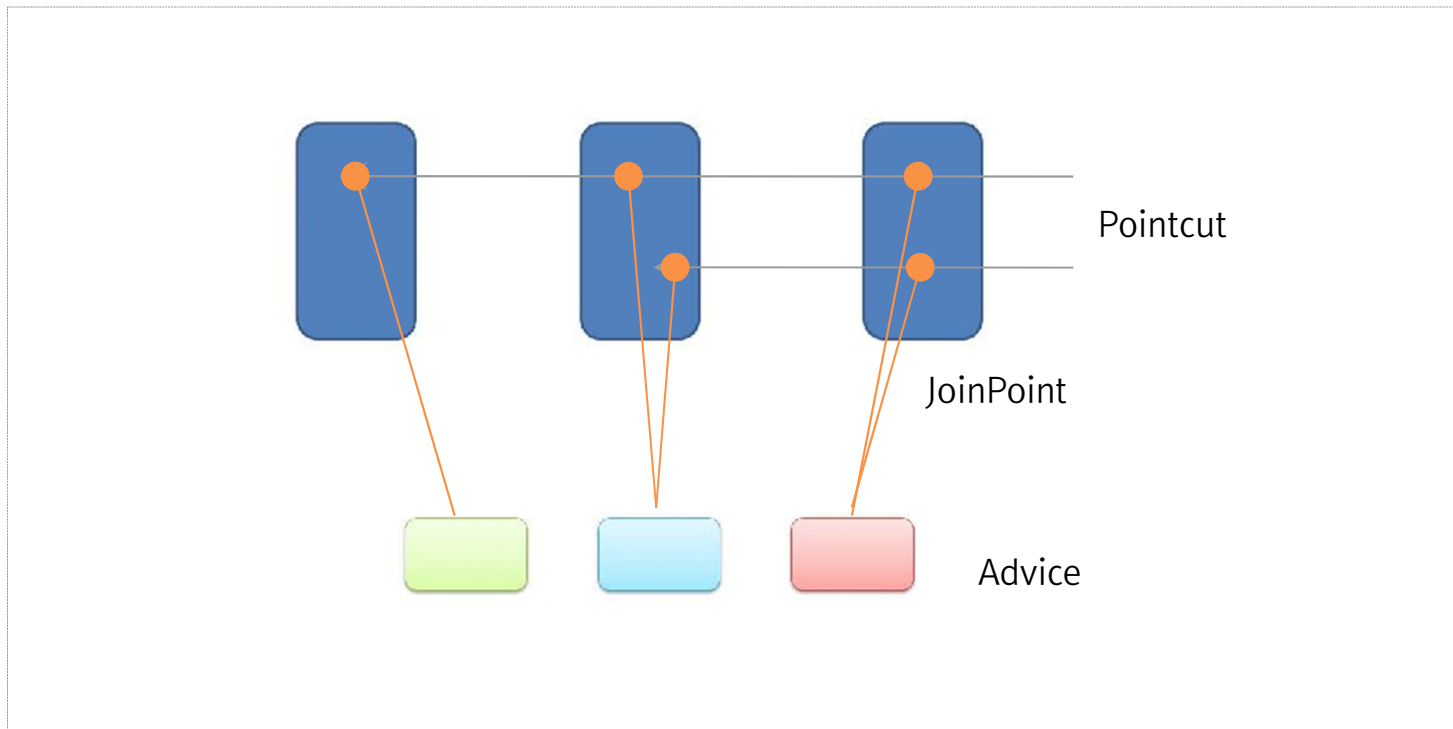
- Advice는 관점(Aspect)의 실제 구현체로 결합점에 삽입되어 동작할 수 있는 코드이다
- Advice 는 결합점(JoinPoint)과 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다
- 특정 Join point에 실행하는 코드

– Weaving

- Pointcut에 의해서 결정된 JoinPoint에 지정된 Advice를 삽입하는 과정
- Weaving은 AOP가 기존의 Core Concerns 모듈의 코드에 전혀 영향을 주지 않으면서 필요한 Crosscutting Concerns 기능을 추가할 수 있게 해주는 핵심적인 처리 과정임

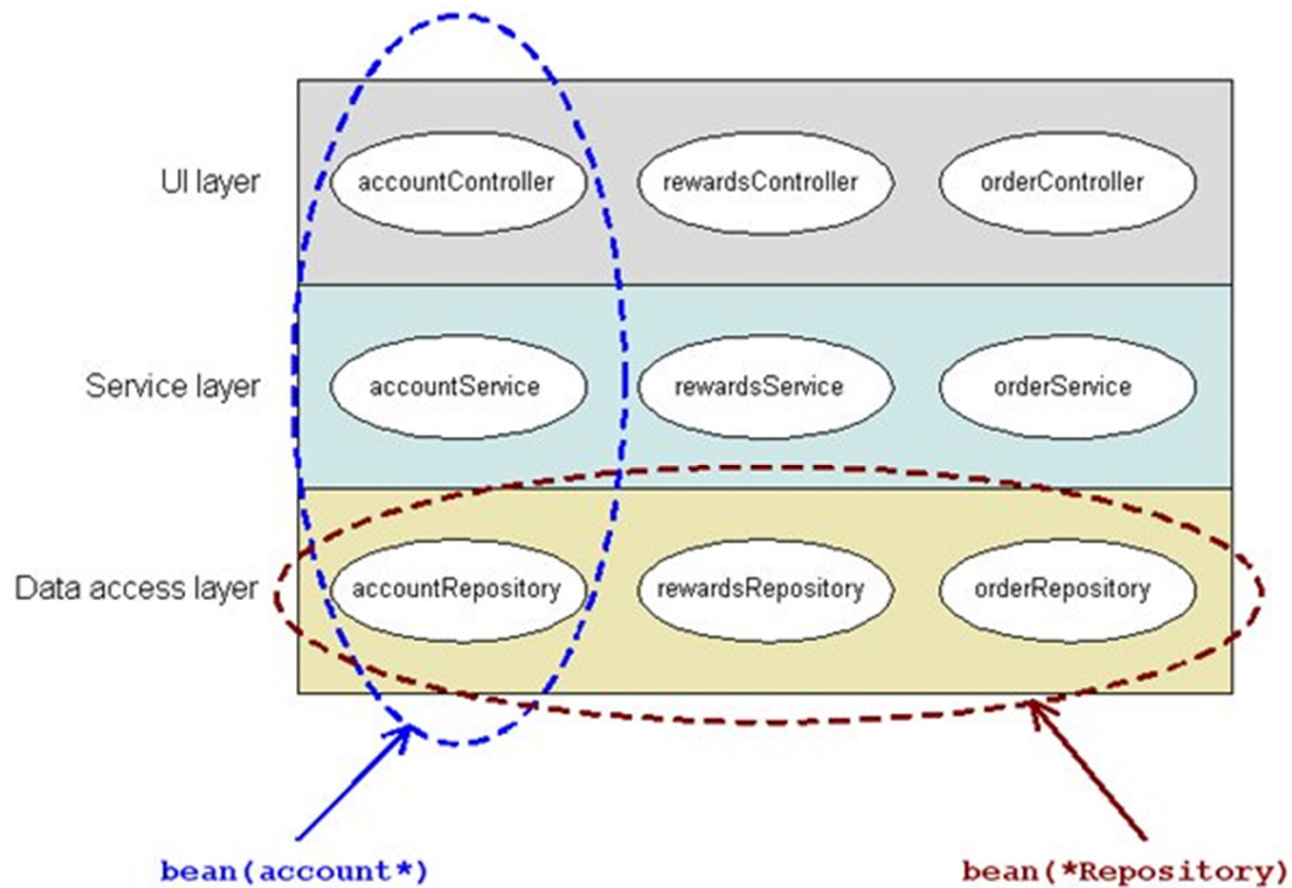
□ 주요 개념

- 조인포인트(JoinPoint), 포인트컷(Pointcut), 어드바이스(Advice)



□ 주요 개념

- 포인트컷(PointCut) 예제 : bean() Pointcut을 이용하여 종적 및 횡적으로 빈을 선택



□ 주요 개념

– Weaving 방식

- 컴파일 시 엮기 : 별도 컴파일러를 통해 핵심 관심사 모듈의 사이 사이에 관점(Aspect) 형태로 만들어진 횡단 관심사 코드들이 삽입되어 관점(Aspect)이 적용된 최종 바이너리가 만들어지는 방식이다. (ex. AspectJ, ...)
- 클래스 로딩 시 엮기 : 별도의 Agent를 이용하여 JVM이 클래스를 로딩할 때 해당 클래스의 바이너리 정보를 변경한다. 즉, Agent가 횡단 관심사 코드가 삽입된 바이너리 코드를 제공함으로써 AOP를 지원하게 된다. (ex. AspectWerkz, ...)
- **런타임 시 엮기** : 소스 코드나 바이너리 파일의 변경 없이 프록시를 이용하여 AOP를 지원하는 방식이다. 프록시를 통해 핵심 관심사를 구현한 객체에 접근하게 되는데, 프록시는 핵심 관심사 실행 전후에 횡단 관심사를 실행한다. 따라서 프록시 기반의 런타임 엮기의 경우 메소드 호출 시에만 AOP를 적용할 수 있다는 제한점이 있다. (ex. Spring AOP, ...)

– Advice 결합점 결합 타입

- Before advice: joinpoint 전에 수행되는 advice
- After returning advice: joinpoint가 성공적으로 리턴된 후에 동작하는 advice
- After throwing advice: 예외가 발생하여 joinpoint가 빠져나갈때 수행되는 advice
- After advice: join point를 빠져나가는(정상적이거나 예외적인 반환) 방법에 상관없이 수행되는 advice
- Around advice: joinpoint 전, 후에 수행되는 advice

□ 주요 기능

- 횡단 관심(CrossCutting Concern) 모듈이 삽입되어 동작할 수 있도록 지정하는 JoinPoint 기능
- 횡단 관심 모듈을 특정 JoinPoint에 사용할 수 있도록 지정하는 Pointcut 기능
- Pointcut 지정을 위한 패턴 매칭 표현식
- Pointcut에서 수행해야하는 동작을 지정하는 Advice 기능
- Pointcut에 의해서 결정된 JoinPoint에 지정된 Advice를 삽입하여 실제 AOP 방식으로 동작

□ 장점

- 중복 코드의 제거
 - 횡단 관심(CrossCutting Concerns)을 여러 모듈에 반복적으로 기술되는 현상을 방지
- 비즈니스 로직의 가독성 향상
 - 핵심기능 코드로부터 횡단 관심 코드를 분리함으로써 비즈니스 로직의 가독성 향상
- 생산성 향상
 - 비즈니스 로직의 독립으로 인한 개발의 집중력을 높임
- 재사용성 향상
 - 횡단 관심 코드는 여러 모듈에서 재사용될 수 있음
- 변경 용이성 증대
 - 횡단 관심 코드가 하나의 모듈로 관리되기 때문에 이에 대한 변경 발생시 용이하게 수행할 수 있음

□ Spring의 AOP 지원

- 스프링은 프록시 기반의 런타임 Weaving 방식을 지원한다.
- 스프링은 AOP 구현을 위해 다음 세가지 방식을 제공한다.
 - @AspectJ 어노테이션을 이용한 AOP 구현
 - XML Schema를 이용한 AOP 구현
 - 스프링 API를 이용한 AOP 구현
- 표준프레임워크 실행환경은 XML Schema를 이용한 AOP 구현 방법을 사용한다.

□ XML 스키마를 이용한 AOP 지원 (2/11)

- Aspect 정의하기

Advice
정의

```
<bean id="adviceUsingXML" class="egovframework.rte.fdl.aop.sample.AdviceUsingXML" />
```

```
<aop:config>
```

```
<aop:pointcut id="targetMethod"
```

```
expression="execution(* egovframework.rte.fdl.aop.sample.*Sample.*(..))" />
```

```
<aop:aspect ref="adviceUsingXML">
```

```
<aop:before pointcut-ref="targetMethod" method="beforeTargetMethod" />
```

```
<aop:after-returning pointcut-ref="targetMethod"
```

```
method="afterReturningTargetMethod" returning="retVal" />
```

```
<aop:after-throwing pointcut-ref="targetMethod"
```

```
method="afterThrowingTargetMethod" throwing="exception" />
```

```
<aop:after pointcut-ref="targetMethod" method="afterTargetMethod" />
```

```
<aop:around pointcut-ref="targetMethod" method="aroundTargetMethod" />
```

```
</aop:aspect>
```

```
</aop:config>
```

PointCut
정의JoinPoint
정의Aspect
정의

❑ XML 스키마를 이용한 AOP 지원(3/11)

- Advice 정의하기 – before advice

```
public class AdviceUsingXML{  
  
    public void beforeTargetMethod(JoinPoint thisJoinPoint){  
        Class clazz = thisJoinPoint.getTarget().getClass();  
        String className = thisJoinPoint.getTarget().getClass().getSimpleName();  
        String methodName = thisJoinPoint.getSignature().getName();  
  
        // 대상 메서드에 대한 로거를 얻어 해당 로거로 현재 class, method 정보 로깅  
        Log logger = LogFactory.getLog(clazz);  
        logger.debug(className + "." + methodName + " executed.");  
  
    }  
    ...  
}
```

❑ XML 스키마를 이용한 AOP 지원(4/11)

- Advice 정의하기 – After returning advice
 - After returning advice는 정상적으로 메소드가 실행될 때 수행된다.

```
public class AdviceUsingXML {  
  
    public void afterReturningTargetMethod(JoinPoint thisJoinPoint,  
        Object retVal) {  
        System.out.println("AspectUsingAnnotation.afterReturningTargetMethod executed." +  
            "return value is [" + retVal + "]);  
    }  
    ...  
}
```

❑ XML 스키마를 이용한 AOP 지원(5/11)

- Advice 정의하기 – After throwing advice
 - After throwing advice 는 메소드가 수행 중 예외사항을 반환하고 종료하는 경우 수행된다.

```
public class AdviceUsingXML {  
    ...  
    public void afterThrowingTargetMethod(JoinPoint thisJoinPoint,  
        Exception exception) throws Exception {  
        System.out.println("AdviceUsingXML.afterThrowingTargetMethod executed.");  
        System.err.println("에러가 발생했습니다.", exception);  
        throw new BizException("에러가 발생했습니다.", exception);  
    }  
    ...  
}
```

❑ XML 스키마를 이용한 AOP 지원(6/11)

– Advice 정의하기 – After (finally) advice

- After (finally) advice 는 메소드 수행 후 무조건 수행된다.
- After advice 는 정상 종료와 예외 발생 경우를 모두 처리해야 하는 경우에 사용된다 (예:리소스 해제 작업)

```
public class AdviceUsingXML {  
  
    public void afterTargetMethod(JoinPoint thisJoinPoint) {  
        System.out.println("AspectUsingAnnotation.afterTargetMethod executed.");  
    }  
    ...  
}
```


❑ XML 스키마를 이용한 AOP 지원(7/11)

– Advice 정의하기 – Around advice

- Around advice 는 메소드 수행 전후에 수행된다.
- Return값을 가공하기 위해서는 Around를 사용해야한다.

```
public class AdviceUsingXML {  
  
    public Object aroundTargetMethod(ProceedingJoinPoint thisJoinPoint)  
        throws Throwable {  
        System.out.println("AspectUsingAnnotation.aroundTargetMethod start.");  
        long time1 = System.currentTimeMillis();  
        Object retVal = thisJoinPoint.proceed();  
  
        System.out.println("ProceedingJoinPoint executed. return value is [" + retVal + "]);  
  
        retVal = retVal + "(modified)";  
        System.out.println("return value modified to [" + retVal + "]);  
  
        long time2 = System.currentTimeMillis();  
        System.out.println("AspectUsingAnnotation.aroundTargetMethod end. Time(" +  
            + (time2 - time1) + ")");  
        return retVal;  
    }  
    ...  
}
```

□ XML 스키마를 이용한 AOP 지원(8/11)

- Aspect 실행하기 – 정상 실행의 경우

```
public class AnnotationAspectTest {  
  
    @Resource(name = "adviceSample")  
    AdviceSample adviceSample;  
  
    @Test  
    public void testAdvice () throws Exception {  
  
        SampleVO vo = new SampleVO();  
  
        String resultStr = annotationAdviceSample.someMethod(vo);  
  
        assertEquals("someMethod executed.(modified)", resultStr);  
    }  
}
```

□ XML 스키마를 이용한 AOP 지원(9/11)

- Aspect 실행하기 – 정상 실행인 경우
 - 콘솔 로그 출력 Advice 적용 순서
 - 1.before
 - 2.around (대상 메소드 수행 전)
 - 3.대상 메소드
 - 4.around (대상 메소드 수행 후)
 - 5.after(finally)
 - 6.after-returning

□ XML 스키마를 이용한 AOP 지원(10/11)

- Aspect 실행하기 – 예외 발생의 경우

```
public class AnnotationAspectTest {  
  
    @Resource(name = "adviceSample")  
    AdviceSample adviceSample;  
  
    @Test  
    public void testAdviceWithException() throws Exception {  
  
        SampleVO vo = new SampleVO();  
        // exception 을 발생하도록 플래그 설정  
        vo.setForceException(true);  
  
        ..  
        try {  
            String resultStr = annotationAdviceSample.someMethod(vo);  
  
            fail("exception을 강제로 발생시켜 이 라인이 수행될 수 없습니다.");  
        } catch (Exception e) {  
            ...  
        }  
    }  
}
```

□ XML 스키마를 이용한 AOP 지원(11/11)

- Aspect 실행하기 – 예외 발생의 경우
 - 콘솔 로그 출력 Advice 적용 순서
 - 1.before
 - 2.around (대상 메소드 수행 전)
 - 3.대상 메소드 (ArithmeticException 예외가 발생한다)
 - 4.after(finally)
 - 5.afterThrowing

□ Pointcut 지정자

- **execution**: 메소드 실행 결합점(join points)과 일치시키는데 사용된다.
- **within**: 특정 타입에 속하는 결합점을 정의한다.
- **this**: 빈 참조가 주어진 타입의 인스턴스를 갖는 결합점을 정의한다.
- **target**: 대상 객체가 주어진 타입을 갖는 결합점을 정의한다.
- **args**: 인자가 주어진 타입의 인스턴스인 결합점을 정의한다.

□ Pointcut 표현식 조합

- '&&' : anyPublicOperation() && inTrading()
- '||' : bean(*dataSource) || bean(*DataSource)
- '!' : !bean(accountRepository)

□ Pointcut 정의 예제

Pointcut	선택된 Joinpoints
execution(public **(..))	public 메소드 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* com.xyz.service.AccountService.*(..))	AccountService 인터페이스의 모든 메소드 실행
execution(* com.xyz.service.*.*(..))	service 패키지의 모든 메소드 실행
execution(* com.xyz.service..*.*(..))	service 패키지와 하위 패키지의 모든 메소드 실행
within(com.xyz.service.*)	service 패키지 내의 모든 결합점
within(com.xyz.service..*)	service 패키지 및 하위 패키지의 모든 결합점
this(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 프록시 개체의 모든 결합점
target(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
args(java.io.Serializable)	하나의 파라미터를 갖고 전달된 인자가 Serializable인 모든 결합점
bean(accountRepository)	“accountRepository” 빈
!bean(accountRepository)	“accountRepository” 빈을 제외한 모든 빈
bean(*)	모든 빈
bean(account*)	이름이 'account'로 시작되는 모든 빈
bean(*Repository)	이름이 “Repository”로 끝나는 모든 빈
bean(accounting/*)	이름이 “accounting/”로 시작하는 모든 빈
bean(*dataSource) bean(*DataSource)	이름이 “dataSource” 나 “DataSource” 으로 끝나는 모든 빈

❑ 실행환경 AOP 가이드라인

- 실행환경은 예외 처리와 트랜잭션 처리에 AOP를 적용함

❑ 실행환경 AOP 가이드라인- 예외 처리(1/2)

- 관점(Aspect) 정의: resources/egovframework.spring/context-aspect.xml

```
<bean id="exceptionTransfer" class="egovframework.rte.fdl.cmmn.aspect.ExceptionTransfer">
...
</bean>

<aop:config>
  <aop:pointcut id="serviceMethod"
    expression="execution(* egovframework.rte.sample..impl.*Impl.*(..))" />

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

Advice 정의

Pointcut
정의JoinPoint
정의

Aspect 정의

❑ 실행환경 AOP 가이드라인- 예외 처리(2/2)

- Advice 클래스 : egovframework.rte.fdl.cmmn.aspect.ExceptionTransfer

```
public class ExceptionTransfer {  
  
    public void transfer(JoinPoint thisJoinPoint, Exception exception) throws Exception {  
        ...  
        if (exception instanceof EgovBizException) {  
            log.debug("Exception case :: EgovBizException ");  
            EgovBizException be = (EgovBizException) exception;  
            getLog(clazz).error(be.getMessage(), be.getCause());  
            processHandling(clazz, exception, pm, exceptionHandlerServices, false);  
            throw be;  
        } else if (exception instanceof RuntimeException) {  
            log.debug("RuntimeException case :: RuntimeException ");  
            RuntimeException be = (RuntimeException) exception;  
            getLog(clazz).error(be.getMessage(), be.getCause());  
            processHandling(clazz, exception, pm, exceptionHandlerServices, true);  
            ...  
            throw be;  
        } else if (exception instanceof FdlException) {  
            ...  
            throw fe;  
        } else {  
            ...  
            throw processException(clazz, "fail.common.msg", new String[] {}, exception, locale);  
        }  
    }  
}
```

□ 실행 환경 AOP 가이드라인- 트랜잭션 처리

- 관점(Aspect) 정의: resources/egovframework.spring/context-transaction.xml

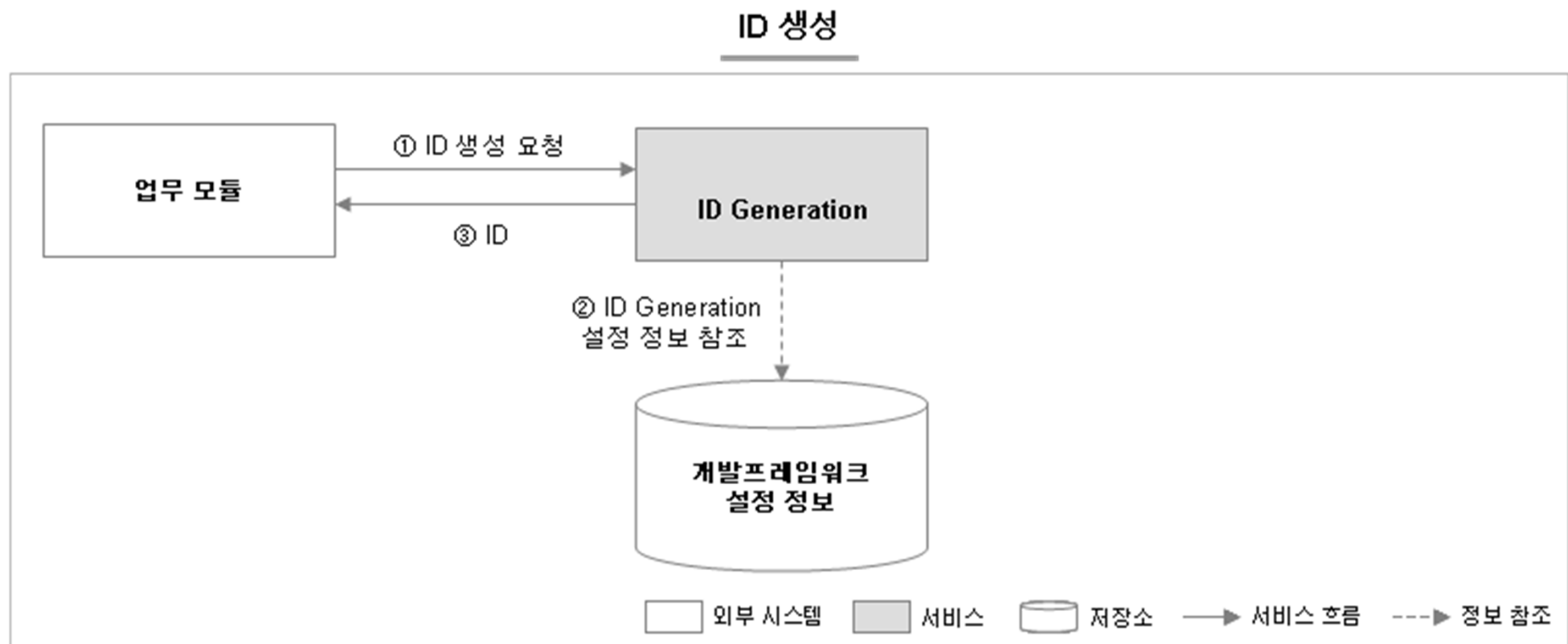
```
<!-- 트랜잭션 관리자를 설정한다. -->
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>

<!-- 트랜잭션 Advice를 설정한다. -->
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="*" rollback-for="Exception"/>
  </tx:attributes>
</tx:advice>

<!-- 트랜잭션 Pointcut를 설정한다. --->
<aop:config>
  <aop:pointcut id="requiredTx"
    expression="execution(* egovframework.rte.sample..impl.*Impl.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="requiredTx" />
</aop:config>
```

□ 서비스 개요

- 다양한 형식의 ID 구조 및 다양한 방식의 ID 생성 알고리즘을 제공하여 시스템에서 사용하는 ID(Identifier)를 생성하는 서비스



□ 주요 기능

- UUID(Universal Unique Identifier) 생성
: UUID(Universal Unique Identifier)를 생성한다.
- Sequence ID 생성
: 순차적으로 증가 또는 감소하는 Sequence ID를 생성한다. 시스템에서는 다수의 Sequence ID가 사용되므로, 각각의 Sequence ID는 구별된다. 시스템의 재시작 시에도 Sequence ID는 마지막 생성된 ID의 다음 ID를 생성한다.
 - Sequence ID 생성
: DB의 SEQUENCE를 활용하여 ID를 생성한다.
 - Table ID 생성
: 키제공을 위한 테이블을 지정하여 ID를 생성한다.

□ UUID(Universally Unique Identifier)란

- UUID는 OSF(Open Software Foundation)에 의해 제정된 고유식별자(Identifier)에 대한 표준이다. UID는 16-byte (128-bit)의 숫자로 구성된다. UUID를 표현하는 방식에 대한 특별한 규정은 없으나, 일반적으로 16진법으로 8-4-4-4-12 형식으로 표현한다.

550e8400-e29b-41d4-a716-446655440000

UUID는 다음 5개의 Version이 존재한다.

- Version 1 (MAC Address)
UUID를 생성시키는 컴퓨터의 MAC 어드레스와 시간 정보를 이용하여 UUID를 생성한다. 컴퓨터의 MAC 어드레스를 이용하므로 어떤 컴퓨터에서 생성했는지 정보가 남기 때문에 보안에 문제가 있다.
- Version 2 (DCE Security)
POSIX UID를 이용하여 UUID를 생성한다.
- Version 3 (MD5 Hash)
URL로부터 MD5를 이용하여 UUID를 생성한다.
- Version 4 (Random)
Random Number를 이용하여 UUID를 생성한다.
- Version 5 (SHA-1 Hash)
SHA-1 Hashing을 이용하여 UUID를 생성한다.

□ 개요

- 새로운 ID를 생성하기 위해 UUID 생성 알고리즘을 이용하여 16 바이트 길이의 ID를 생성한다. String 타입의 ID 생성과 BigDecimal 타입의 ID 생성을 지원한다. 지원하는 방법은 설정에 따라서 Mac Address Base Service , IP Address Base Service , No Address Base Service 세가지 유형이 있다.

□ Mac Address Base Service(1/2)

- MAC Address를 기반으로 유일한 Id를 생성하는 UUldGenerationService
- 설정

```
<bean name="UUldGenerationService"
  class="egovframework.rte.fdl.idgnr.impl.EgovUUldGnrService">
  <property name="address">
    <value>00:00:F0:79:19:5B</value>
  </property>
</bean>
```

❑ Mac Address Base Service(2/2)

– Sample

```
@Resource(name="UUIdGenerationService")
private EgovIdGnrService uUidGenerationService;

@Test
public void testUUIdGeneration() throws Exception {
    assertNotNull(uUidGenerationService.getNextStringId());
    assertNotNull(uUidGenerationService.getNextBigDecimalId());
}
```

❑ IP Address Base Service

- IP Address를 기반으로 유일한 Id를 생성하는 UUldGenerationService
- 설정

```
<bean name="UUldGenerationServiceWithIP"
  class="egovframework.rte.fdl.idgnr.impl.EgovUUldGnrService">
  <property name="address">
    <value>100.128.120.107</value>
  </property>
</bean>
```

- Sample

```
@Resource(name="UUldGenerationServiceWithIP")
private EgovIdGnrService uUldGenerationServiceWithIP;

@Test
public void testUUldGenerationIP() throws Exception {
  assertNotNull(uUldGenerationServiceWithIP.getNextStringId());
  assertNotNull(uUldGenerationServiceWithIP.getNextBigDecimalId());
}
```


❑ No Address Base Service

- IP Address 설정없이 Math.random()을 이용하여 주소정보를 생성하고 유일한 Id를 생성하는 UuidGenerationService

- 설정

```
<bean name="UuidGenerationServiceWithoutAddress"  
      class="egovframework.rte.fdl.idgnr.impl.EgovUuidGnrService"/>
```

- Sample

```
@Resource(name="UuidGenerationServiceWithoutAddress")  
private EgovIdGnrService uuidGenerationServiceWithoutAddress;  
  
@Test  
public void testUuidGenerationNoAddress() throws Exception {  
    assertNotNull(uuidGenerationServiceWithoutAddress.getNextStringId());  
    assertNotNull(uuidGenerationServiceWithoutAddress.getNextBigDecimalId());  
}
```

□ 개요

- 새로운 ID를 생성하기 위해 Database의 SEQUENCE를 사용하는 서비스이다. 서비스를 이용하는 시스템에서 Query를 지정하여 아이디를 생성할 수 있도록 하고 Basic Type Service와 BigDecimal Type Service 두가지를 지원한다.

□ Basic Type Service(1/2)

- 기본타입 ID를 제공하는 서비스로 int, short, byte, long 유형의 ID를 제공한다.
- DB Schema

- CREATE SEQUENCE idstest MINVALUE 0;

```
<bean name="primaryTypeSequencelds"  
  class="egovframework.rte.fdl.idgnr.impl.EgovSequenceldGnrService"  
  destroy-method="destroy">  
  <property name="dataSource" ref="dataSource"/>  
  <property name="query" value="SELECT idstest.NEXTVAL FROM DUAL"/>  
</bean>
```

❑ Basic Type Service(2/2)

– Sample

```
@Resource(name="primaryTypeSequencels")
private EgovIdGnrService primaryTypeSequencels;

@Test
public void testPrimaryTypeIdGeneration() throws Exception {
    //int
    assertNotNull(primaryTypeSequencels.getNextIntegerId());
    //short
    assertNotNull(primaryTypeSequencels.getNextShortId());
    //byte
    assertNotNull(primaryTypeSequencels.getNextByteId());
    //long
    assertNotNull(primaryTypeSequencels.getNextLongId());
}
```

❑ BigDecimal Type Service(1/2)

- BigDecimal ID를 제공하는 서비스로 기본타입 ID 제공 서비스 설정에 추가적으로 useBigDecimals을 “true”로 설정하여 사용하도록 한다.

- DB Schema

```
CREATE SEQUENCE idstest MINVALUE 0;
```

- 설정

```
<bean name="bigDecimalTypeSequenceIds"
  class="egovframework.rte.fdl.idgnr.impl.EgovSequenceIdGnrService"
  destroy-method="destroy">
  <property name="dataSource" ref="dataSource"/>
  <property name="query" value="SELECT idstest.NEXTVAL FROM DUAL"/>
  <property name="useBigDecimals" value="true"/>
</bean>
```

❑ BigDecimal Type Service(2/2)

– Sample

```
@Resource(name="bigDecimalTypeSequencelds")
private EgovIdGnrService bigDecimalTypeSequencelds;

@Test
public void testBigDecimalTypeIdGeneration() throws Exception {
    //BigDecimal
    assertNotNull(bigDecimalTypeSequencelds.getNextBigDecimalId());
}
```

❑ DB 벤더 별 SEQUENCE 및 Query 설정

- DB 벤더 별로 SEQUENCE에 대한 지원 여부 및 사용 방식이 다르므로 이를 고려하여 설정해야 한다. (DB 벤더 별 SEQUENCE 지원 여부 및 설정 방식은 wiki 참조)

□ 개요

- 새로운 아이디를 얻기 위해서 별도의 테이블을 생성하고 키 값과 키 값에 해당하는 아이디 값을 입력하여 관리하는 기능을 제공하는 서비스로 table_name(CHAR 또는 VARCHAR타입), next_id(integer 또는 DECIMAL type) 두 칼럼을 필요로 한다. 별도의 테이블에 설정된 정보만을 사용하여 제공하는 Basic Service와 String ID의 경우에 적용이 가능한 prefix와 채울 문자열 지정이 가능한 Strategy Base Service를 제공한다.

□ Basic Service(1/3)

- 테이블에 지정된 정보에 의해서 아이디를 생성하는 서비스로 사용하고자 하는 시스템에서 테이블을 생성해서 사용할 수 있다.
- DB Schema
 - ID Generation 서비스를 쓰고자 하는 시스템에서 미리 생성해야 할 DB Schema 정보임

```
CREATE TABLE ids ( table_name varchar(16) NOT NULL,  
                    next_id DECIMAL(30) NOT NULL,  
                    PRIMARY KEY (table_name));  
INSERT INTO ids VALUES('id','0');
```

□ Basic Service(2/3)

– 설정

```
<bean name="basicService" class="egovframework.rte.fdl.idgnr.impl.EgovTableIdGnrService"
    destroy-method="destroy">
    <property name="dataSource" ref="dataSource"/>
    <property name="blockSize" value="10"/>
    <property name="table" value="ids"/>
    <property name="tableName" value="id"/>
</bean>
```

- blockSize: Id Generation 내부적으로 사용하는 정보로 ID 요청시마다 DB접속을 하지 않기 위한 정보(지정한 횟수마다 DB 접속 처리)
- table: 생성하는 테이블 정보로 사용처에서 테이블명 변경 가능
- tableName: 사용하고자 하는 아이디 개별 인식을 위한 키 값(대개의 경우는 테이블 별로 아이디가 필요하기에 tableName이라고 지정함)

❑ Basic Service(3/3)

– Sample

```
@Resource(name="basicService")
private EgovIdGnrService basicService;

@Test
public void testBasicService() throws Exception {
    //int
    assertNotNull(basicService.getNextIntegerId());
    //short
    assertNotNull(basicService.getNextShortId());
    //byte
    assertNotNull(basicService.getNextByteId());
    //long
    assertNotNull(basicService.getNextLongId());
    //BigDecimal
    assertNotNull(basicService.getNextBigDecimalId());
    //String
    assertNotNull(basicService.getNextStringId());
}
```


❑ Strategy Base Service(1/3)

- 아이디 생성을 위한 룰을 등록하고 룰에 맞는 아이디를 생성할 수 있도록 지원하는 서비스로 위의 Basic Service에서 추가적으로 Strategy정보 설정을 추가하여 사용 할 수 있다. 단, 이 서비스는 String 타입의 ID만을 제공한다.
- DB Schema

```
CREATE TABLE idttest( table_name varchar(16) NOT NULL,  
    next_id DECIMAL(30) NOT NULL,  
    PRIMARY KEY (table_name));  
INSERT INTO idttest VALUES('test','0');
```

❑ Strategy Base Service(2/3)

– 설정

```
<bean name="Ids-TestWithGenerationStrategy"  
    class="egovframework.rte.fdl.idgnr.impl.EgovTableIdGnrService"  
    destroy-method="destroy">  
    <property name="dataSource" ref="dataSource"/>  
    <property name="strategy" ref="strategy"/>  
    <property name="blockSize" value="1"/>  
    <property name="table" value="idttest"/>  
    <property name="tableName" value="test"/>  
</bean>  
  
<bean name="strategy"  
    class="egovframework.rte.fdl.idgnr.impl.strategy.EgovIdGnrStrategyImpl">  
    <property name="prefix" value="TEST-"/>  
    <property name="ciphers" value="5"/>  
    <property name="fillChar" value="*"/>  
</bean>
```

- strategy: 아래에 정의된 MixPrefix 의 bean name 설정
- prefix: 아이디의 앞에 고정적으로 붙이고자 하는 설정값 지정
- ciphers: prefix를 제외한 아이디의 길이 지정
- fillChar: 0을 대신하여 표현되는 문자

❑ Strategy Base Service(3/3)

– Sample

```
@Resource(name="Ids-TestWithGenerationStrategy")
private EgovIdGnrService idsTestWithGenerationStrategy;

@Test
public void testIdGenStrategy() throws Exception {
    initializeNextLongId("test", 1);

    // prefix : TEST-, cipers : 5, fillChar :*)
    for (int i = 0; i < 5; i++) {
        assertEquals("TEST-****" + (i + 1),
            idsTestWithGenerationStrategy.getNextStringId());
    }
}
```

□ 서비스 개요

- Logging은 시스템의 개발이나 운용 시 발생할 수 있는 어플리케이션 내부정보에 대해서, 시스템의 외부 저장소에 기록하여, 시스템의 상황을 쉽게 파악할 수 있게 지원하는 서비스
- 표준프레임워크 3.0부터 SLF4J가 적용이 되었으며 Log4j를 함께 사용하였다.

□ 주요 기능

- 로깅 환경 설정 지원
 - 서버 시스템 별 상세한 로그 정책 부여
 - 다양한 형식(날짜 형식, 시간 형식 등)의 로그 메시지 형태 지정
 - 다양한 매체(File, DBMS, Message, Mail 등)에 대한 기록 기능 설정
- 로그 기록
 - 레벨(debug, info, warn, error 등)별로 로그를 기록

□ Logging 서비스

- 시스템의 개발이나 운용시 발생할 수 있는 사항에 대해서, 시스템의 외부 저장소에 기록하여, 시스템의 상황을 쉽게 파악할 수 있음.
- 많은 개발자가 Log를 출력하기 위해 일반적으로 사용하는 방식은 `System.out.println()`임.
하지만 이 방식은 간편한 반면에 다음과 같은 이유로 권장하지 않음.
 - 콘솔 로그를 출력 파일로 리다이렉트 할 지라도, 어플리케이션 서버가 재 시작할 때 파일이 **overwrite**될 수도 있음.
 - 개발/테스팅 시점에만 `System.out.println()`을 사용하고 운영으로 이관하기 전에 삭제하는 것은 좋은 방법이 아님.
 - `System.out.println()` 호출은 디스크 I/O동안 동기화(synchronized)처리가 되므로 시스템의 throughput을 떨어뜨림.
 - 기본적으로 stack trace 결과는 콘솔에 남는다. 하지만 시스템 운영 중 콘솔을 통해 Exception을 추적하는 것은 바람직하지 못함.

□ Log4j 환경 설정하는 방법

- 프로그래밍내에서 직접 설정하는 방법
- 설정 파일을 사용하는 방법

□ 중요 컴포넌트 설명

컴포넌트	설명
Logger	로그의 주체 (로그 파일을 작성하는 클래스) - 설정을 제외한 거의 모든 로깅 기능이 이를 통해 처리 됨. 어플리케이션 별로 사용할 로거(로거명 기반)를 정의하고 이에 대해 로그레벨과 Appender를 지정할 수 있음.
Appender	로그를 출력하는 위치를 의미하며, Log4J API문서의 XXXAppender로 끝나는 클래스들의 이름을 보면, 출력위치를 어느정도 짐작할 수 있음.
Layout	Appender의 출력포맷 - 일자, 시간, 클래스명등 여러가지 정보를 선택하여 로그정보내용으로 지정할 수 있음.

□ 로그레벨 지정하기

- log4j에서는 기본적으로 debug, info, warn, error, fatal의 다섯 가지 로그레벨이 있음
- (ERROR > WARN > INFO > DEBUG > TRACE)

로그 레벨	설명
Error	- 요청을 처리하는중 문제가 발생한 상태를 나타냄
Warn	- 처리 가능한 문제이지만, 향후 시스템 에러의 원인이 될 수 있는 경고성 메시지를 나타냄.
Info	- 로그인, 상태변경과 같은 정보성 메시지를 나타냄.
Debug	- 개발시 디버그 용도로 사용할 메시지를 나타냄.
Trace	- 디버그 레벨이 너무 광범위한 것을 해결하기 위해서 좀 더 상세한 상태를 나타냄.

□ Appender

- log4j 는 콘솔, 파일, DB, socket, message, mail 등 다양한 로그 출력 대상과 방법을 지원하는데, 이에 대해 log4j 의 Appender 로 정의할 수 있다.

Appender	설명
ConsoleAppender	<ul style="list-style-type: none"> - 콘솔화면으로 출력하기 위한 appender임 - org.apache.log4j.ConsoleAppender : Console 화면으로 출력하기 위한 Appender
FileAppender	<ul style="list-style-type: none"> - FileAppender는 로깅을 파일에 하고 싶을 때 사용함.
RollingFileAppender	<ul style="list-style-type: none"> - FileAppender는 지정한 파일에 로그가 계속 남으므로 한 파일의 크기가 지나치게 커질 수 있으며, 계획적인 로그관리가 불가능해짐. - RollingFileAppender는 파일의 크기 또는 파일백업인덱스 등의 지정을 통해서 특정크기 이상 파일의 크기가 커지게 되면 기존파일을 백업파일로 바꾸고, 다시 처음부터 로깅을 시작함
JDBCAppender	<ul style="list-style-type: none"> - DB에 로그를 출력하기 위한 Appender로 하위에 Driver, URL, User, Password, Sql과 같은 parameter를 정의할 수 있음. - 다음은 log4j.xml 파일 내의 JDBCAppender에 대한 속성 정의 내용임. - EgovConnectionFactory 빈 설정이 필요함

Q&A

감사합니다.