전자정부 표준프레임워크

실행환경 (공통기반)

eGovFrame

Contents

eGovFrame

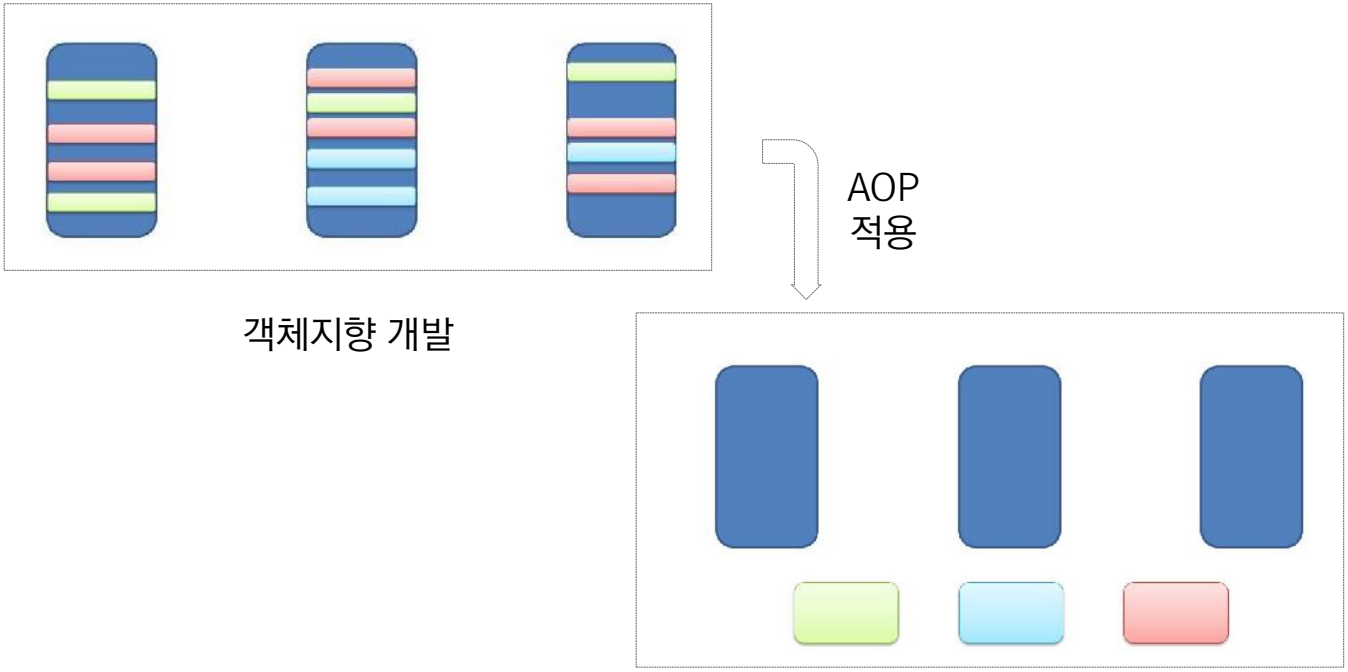
1. 공통기반 레이어

1. AOP
2. ID Generation
3. Logging

eGovFrame

❑ 서비스 개요

- 객체지향 프로그래밍(Object Oriented Programming)을 보완하는 개념으로 어플리케이션을 객체지향적으로 모듈화 하여 작성하더라도 다수의 객체들에 분산되어 중복적으로 존재하는 공통 관심사가 여전히 존재한다. AOP는 이를 횡단관심으로 분리하여 핵심관심과 엮어서 처리할 수 있는 방법을 제공한다.
- 로깅, 보안, 트랜잭션 등의 공통적인 기능의 활용을 기존의 비즈니스 로직에 영향을 주지 않고 모듈화 처리를 지원하는 프로그래밍 기법



❑ 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

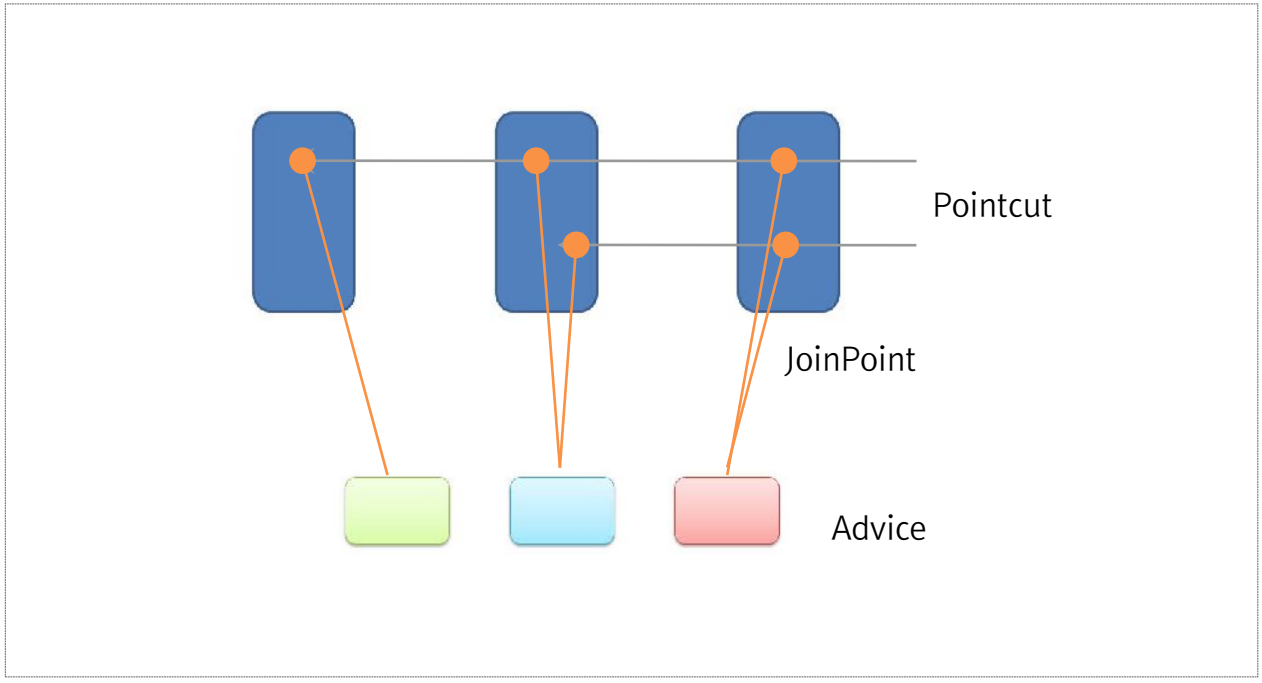
- Advice는 관점(Aspect)의 실제 구현체로 결합점에 삽입되어 동작할 수 있는 코드이다
- Advice 는 결합점(JoinPoint)과 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다
- **특정 Join point에 실행하는 코드**

– 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

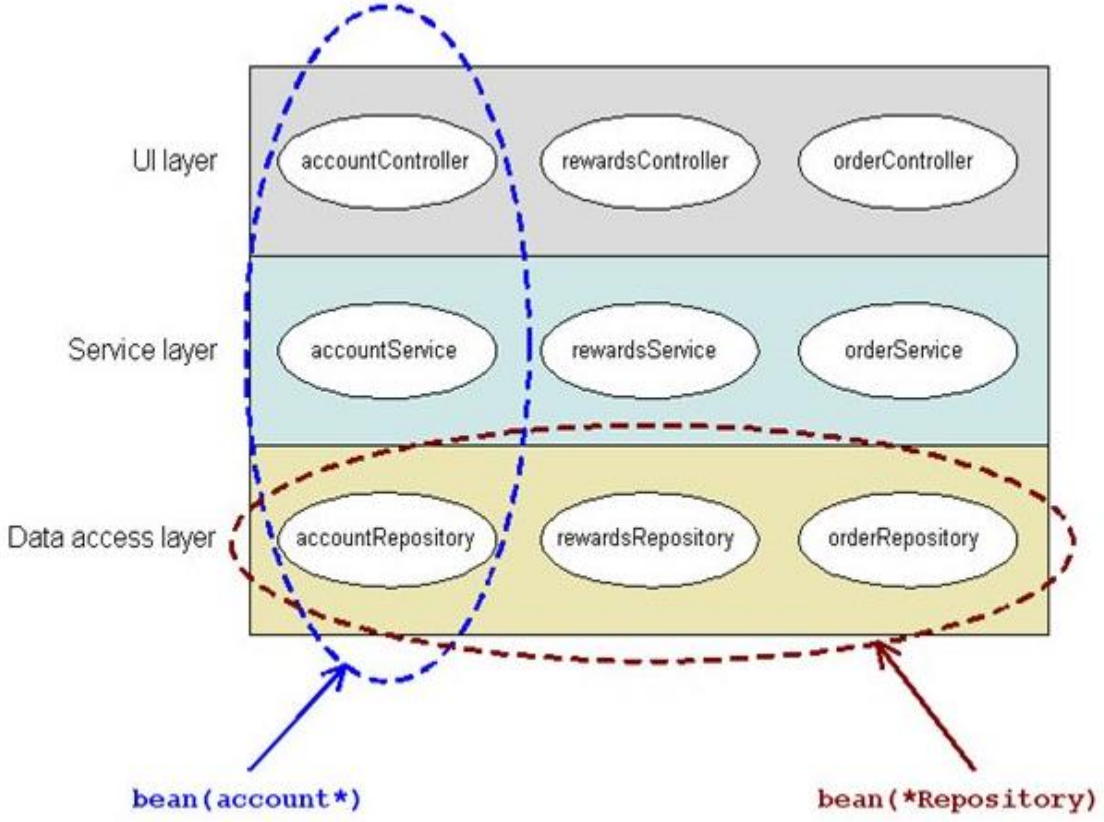
❑ 주요 개념

- 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 JointPoint 기능
- 횡단 관심 모듈을 특정 JointPoint에 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 방식대로 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

JoinPoint 정의

PointCut 정의

Aspect 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>
```

❑ 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 returing 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.after-returning
    5.after(finally)
    6.around (대상 메소드 수행 후)

❑ 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.afterThrowing
    5.after(finally)

❑ 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 처리**에 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 정의

JoinPoint 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);
    }
  }
}
```

❑ 실행환경 AOP 가이드라인– 트랜잭션 처리

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

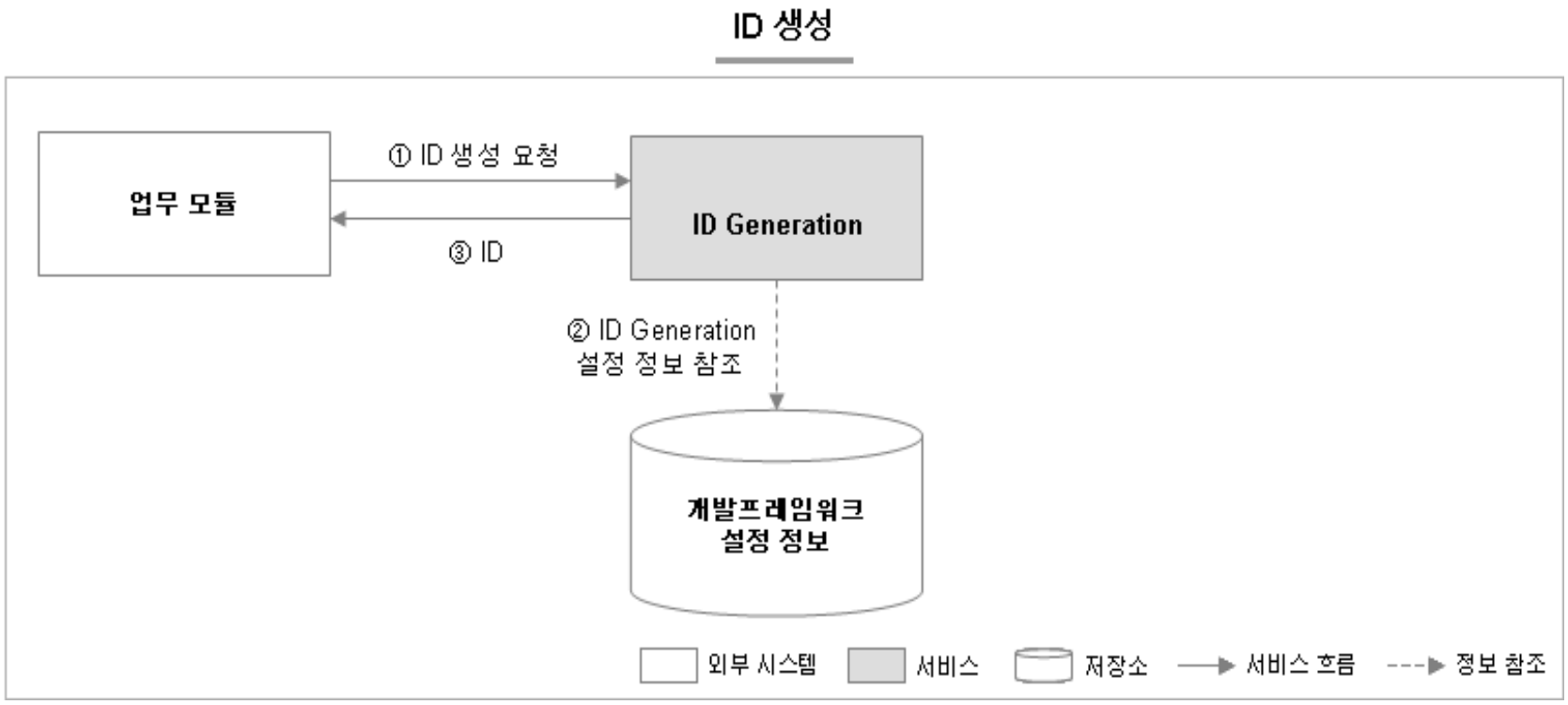
```
〈!-- 트랜잭션 관리자를 설정한다. --〉
〈bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"〉
 〈property name="dataSource" ref="dataSource"/〉
〈/bean〉

〈!-- 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 UUIdGenerationService
- 설정

```
<bean name="UUIdGenerationService"
    class="egovframework.rte.fdl.idgnr.impl.EgovUUIdGnrService">
  <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 UUIdGenerationService
- 설정

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

- Sample

```
@Resource(name="UUIdGenerationServiceWithIP")
private EgovIdGnrService uUIdGenerationServiceWithIP;

@Test
public void testUUIdGenerationIP() throws Exception {
  assertNotNull(uUIdGenerationServiceWithIP.getNextStringId());
  assertNotNull(uUIdGenerationServiceWithIP.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;
```

- 설정

```
<bean name="primaryTypeSequenceIds“
      class="egovframework.rte.fdl.idgnr.impl.EgovSequenceIdGnrService“
      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="primaryTypeSequenceIds")
private EgovIdGnrService primaryTypeSequenceIds;

@Test
public void testPrimaryTypeIdGeneration() throws Exception {
  //int
  assertNotNull(primaryTypeSequenceIds.getNextIntegerId());
  //short
  assertNotNull(primaryTypeSequenceIds.getNextShortId());
  //byte
  assertNotNull(primaryTypeSequenceIds.getNextByteId());
  //long
  assertNotNull(primaryTypeSequenceIds.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="bigDecimalTypeSequenceIds")
private EgovIdGnrService bigDecimalTypeSequenceIds;

@Test
public void testBigDecimalTypeIdGeneration() throws Exception {
  //BigDecimal
  assertNotNull(bigDecimalTypeSequenceIds.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 횟수 마다 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="cipers" value="5"/>
    <property name="fillChar" value="*"/>
</bean>
```

- strategy: 아래에 정의된 MixPrefix 의 bean name 설정
- prefix: 아이디의 앞에 고정적으로 붙이고자 하는 설정값 지정
- cipers: 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**
  - 서브 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 Log을 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 overwrite될 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 처리됨. 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	- 콘솔화면으로 출력하기 위한 appender임 - org.apache.log4j.ConsoleAppender : Console 화면으로 출력하기 위한 Appender
FileAppender	- FileAppender는 로깅을 파일에 하고 싶을 때 사용함.
RollingFileAppender	- 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	- 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.