JEUS JMS 안내서

Document info
Document name: JEUS JMS 안내서
Document date: 2005-06-06
Manual release version: 3
Software Version: JEUS 5

차 례

그림 목차

표 목차

매뉴얼에관해서

매뉴얼의 대상

이 문서는 JEUS JMS 를 사용하고 유지보수 임무를 맡은 사람들과 시스템 관리자들이 읽어야 한다.

매뉴얼의 전제 조건

이 매뉴얼을 읽기 위한 2 가지 조건이 있다.

1. 첫째, JEUS Server 안내서메뉴얼을 읽고 이해를 해야 한다.
2. 둘째, 일반적인 JMS 개발에 관한 지식을 가지고 있어야 한다. 이런 지식이 부족할 경우 JMS 기술과 관련된 문서를 읽어야 하고 스펙 문서를 참조 하기 바란다(http://java.sun.com 을 참조 한다).

주의: 이 매뉴얼에서는 J2EE 나 JMS 에 대한 기본적인 내용은 설명하지 않는다.

이 문서의 구성

이 매뉴얼은 8 장의 본문과 2 장의 부록으로 나누어져 있다.

8 장의 본문 내용은 다음과 같다.

1. 소개: 인사말과 개요
2. 따라하기: 예제를 가지고 단계별로 시작하기
3. JEUS JMS 의 개요: JEUS JMS 엔진의 기본적인 개념과 구성
4. JMS Server 구성: JMS Server 의 구성과 그 하위 항목들
5. JMS Server 관리: JMS Server 의 하위 항목들의 추가와 제거
6. JMS Server 모니터링: JMS Server 상태 보기

7. JEUS JMS 프로그래밍: JMS 클라이언트 어플리케이션의 개발

8. 결론

부록:

A. jmsadmin Console Tool Reference: jmsadmin 콘솔 툴에 대한 설명

B. JMSMain.xml Xml Configuration Reference: Web container 의 WEBMain.xml 의 환경파일을 참조하여 완성

관련 문서들

관련 문서들:

- J2EE 1.4 스펙
- JMS 1.1 스펙
- JEUS Server 안내서 (특히 transaction 관련 본문)

일러두기

표기 예	내용
텍스트	본문, 12 포인트, 바탕체 Times New Roman
텍스트	본문 강조
CTRL+C	Ctrl 와 동시에 C 를 누름
`public class myClass { }`	Java 코드
`<system-config>`	XML 문서

표기 예	내용
참조: / 주의:	참조 사항과 주의할 사항
Configuration 메뉴를 연다	GUI 의 버튼 같은 컴포넌트
JEUS_HOME	JEUS 가 실제로 설치된 디렉토리 예) c:\jeus50
`jeusadmin nodename`	콘솔 명령어와 문법
`[ command argument ]`	옵션 파라미터
< xyz >	'<'와 '>' 사이의 내용이 실제 값으로 변경됨
|	선택 사항. 예) A|B: A 나 B 중 하나
…	파라미터 등이 반복되어서 나옴
?, +, *	보통 XML 문서에 각각 "없거나, 한 번", "한 번 이상", "없거나, 여러 번"을 나타낸다.
. . .	XML 이나 코드등의 생략
<<FileName.ext>>	코드의 파일명
그림 1.	그림 이름 또는 표 이름

OS 에 대해서

본 문서는 모든 예제와 환경 구성을 Microsoft Windows™의 스타일을 따랐다. 유닉스와 같은 다른 환경에서 작업하는 사람은 몇가지 사항만 고려하면

별 무리없이 사용할 수 있다. 대표적인 것이 디렉토리 구분자인데, Windows 스타일인 “\”를 유닉스 스타일인 “/”로 바꿔서 사용하면 무리가 없다. 이외에 환경 변수도 유닉스 스타일로 변경해서 사용하면 된다.

그러나 Java 표준을 고려해서 문서를 작성했기 때문에, 대부분의 내용은 동일하게 적용된다.

용어 설명

다음에 소개되는 용어는 본 문서 전체에 걸쳐서 사용되는 용어이다. 용어가 이해하기 어렵거나 명확하지 않을 때는 아래 정의를 참조하기 바란다.

용어	정의
Connection Factory	JMS 클라이언트가 connection 을 만들기 위해서 사용하는 관리자 객체 이다. Application 에서는 이 클래스의 객체를 JNDI 를 통해서 사용할 수 있다.
Connection	Connection 객체는 JEUS JMS Server 와의 연결를 의미한다. Connection 객체는 주로 Session 을 만들기 위해서 사용한다.
Destination	Destination 객체는 queue 나 topic 을 나타내는 객체로 JEUS JMS Server 에 지정되어 있는 queue 나 topic 의 정보를 가지고 있다. Application 에서는 이 객체를 JNDI 를 통해서 사용할 수 있다.
Durable Subscriber	동적으로 JMS Server 가 메시지를 보관하고 있는 topic destination 에 대해 subscription 을 등록 한다.
Fail over	서버에 장애가 발생 하더라도 서비스가 중단되지 않고 계속해서 운영이 가능하도록 한다 .
Filtering	어떤 조건에 맞는 특정한 메시지만을 선택하는 용어 이다.
JMS	어플리케이션이 메시지를 만들고, 보내고, 받는 것을 제공하는 J2EE 서비스 이다.

용어	정의
	것을 제공하는 J2EE 서비스 이다.
JMS client	JMS API 을 통해서 메시지를 보내고 받는 클라이언트 어플리케이션을 말한다.
JMS engine	JMS Server 으로부터 메시지 서비스를 제공하는 컨테이너 내에 등록된 엔진을 말한다.
JMS server	메시지 서비스를 처리하는 JMS 엔진이 만든 서버 프로그램을 말한다.
Message	JMS 클라이언트 사이에서 통신하는 정보의 단위를 말한다.
Message consumer	destination 으로 메시지를 받기 위한 인터페이스를 제공하고 session 에서 만들어지는 객체이다.
Message listener	Message consumer 가 메시지를 기다리기 위해서 등록하는 객체이다.
Message producer	Destination 으로 메시지를 보내기 위한 인터페이스를 제공하고 session 에서 만들어지는 객체이다.
Message selector	보내어진 메시지들 중에서 메시지를 선택하는 조건을 말한다.
Peer-to-Peer	하나의 클라이언트에서 하나의 클라이언트로 직접 통신하는 것을 말한다.
Pub	Publishing 에 대한 축약된 표현이다. 이것은 topic destination 으로 메시지를 보내는 것을 의미한다.
Queue	하나의 message consumer 을 받아 들이고 메시지를 보관하는 destinatin 을 말하다.

용어	정의
Session	메시지를 보내고 받는데 사용되는 쓰레드 형태의 context 를 말하다.
Sub	subscribing 의 축약어로서, topic destination 으로부터 메시지를 받아들이는 것을 의미한다.
Thread pool	실행 후의 쓰레드를 보관하는 장소이고 실행하기 위해서 기다리는 장소이다.
Topic	하나 이상의 message consumer 를 받아 들이고 메시지를 보관하는 destination 을 말하다.
Two-phase commit	여러 개의 리소스 매니저를 사용하는 환경에서 commit 을 보장하는 프르토콜을 말한다.

연락처

Korea
Tmax Soft Co., Ltd
18F Glass Tower, 946-1, Daechi-Dong, Kangnam-Gu, Seoul 135-708
South Korea
Tel: 82-2-6288-2114
Fax: 82-2-6288-2115
Email: info@tmax.co.kr
Web (Korean): http://www.tmax.co.kr

USA
Tmax Soft, Inc.
560 Sylvan Ave, Englewood Cliffs NJ 07632
USA
Tel: 1-201-567-8266
FAX: 1-201-567-7339
Email: info@tmaxsoft.com
Web (English): http://www.tmaxsoft.com

Japan
Tmax Soft Japan Co., Ltd.
6-7 Sanbancho, Chiyoda-ku, Tokyo 102-0075
Japan
Tel: 81-3-5210-9270
FAX: 81-3-5210-9277
Email: info@tmaxsoft.co.jp
Web (Japanese): http://www.tmaxsoft.co.jp

China
Beijing Silver Tower, RM 1507, 2# North Rd Dong San Huan,
Chaoyang District, Beijing, China, 100027
Tel: 86-10-6410-6148
Fax: 86-10-6410-6144
E-mail : info@tmaxchina.com.cn
Web (Chinese): http://www.tmaxchina.com.cn

1 소개

메시지 기반 미들웨어**(MOM)**는 현재 가장 보편화된 형태의 미들웨어 중의 한가지로써 엔터프라이즈 시스템간에 이루어지는 정보 교환의 양상을 메시지라는 형태의 데이터와 데스티네이션 이라는 가상적인 체널로 추상화한다.

즉 사용자가 생성한 메시지는 관리자에 의해 생성돤 가상 채널, 데스티네이션에 전달되며 이 정보가 필요한 어플리케이션은 언제라도 이 데스티네이션에 자신을 등록하여 메시지를 얻어올 수 있다.

위와 같은 비동기적 통신, 메시지 생성자와 소비자 간의 결합성이 적은 장점 외에 **MOM** 이 제공하는 빠르고 안정적인 메시지 전달 방식은 물리적, 상황적으로 거리가 있는 **EIS** 간 통합에 적합하며 실제로 상당한 지지를 받으며 널리 사용되고 있다.

JMS 는 썬 마이크로시스템즈™ 에서 제정한 자바 기반의 **MOM** 표준이다. **JMS** 는 그 자체가 메시지 서비스를 정의하지는 않으며 다만 공통적인 **API** 와 **API** 의 세만틱을 정의하여 다양한 방식의 메시징 서비스가 가능하도록 배려하고 있다. 이러한 벤더에 종속적이지 않은 **JMS** 의 **API** 를 사용함으로써 다양한 하부 메시지 서비스와 관계없는 안정적인 통신 어플리케이션을 구성하는게 가능하게 되었다.

JEUS MQ 는 **JMS 1.1** 스펙을 구현하며 부가적 스펙인 **XA** 트랜잭션을 완전히 지원한다. 영속적인 메시지 서비스를 위한 파일 및 **DB** 스토리지를 사용할 수 있으며 분산된 시스템 채널을 가상적인 하나의 채널로 구성하기 위한 데스티네이션 클러스터링을 지원한다. 또한 대량의 클라이언트에 대한 서비스를 위해하기 위해 **NBIO** 통신 라이브러리를 사용할 수 있다.

이 매뉴얼은 JEUS JMS 모듈의 전체 기능에 대해서 설명한다. 우선 간단하게 사용 방법을 살펴본 후에(2 장), 각 서브 모듈의 설정과 관리하는 방법에 대해서 다룬다(3,4,5,6 장). 그리고 Java 어플리케이션이 엔터프라이즈 메시징 시스템으로 접근하기 위한 방법에 대해서도 다룬다(7 장).

2 따라하기

이 장에서는 JEUS JMS Server 의 시작과 정지, 그리고 간단한 예제들을 실행하는 기본적인 내용을 다룰 것이다.

JEUS WAS 를 인스톨 한 후에 다음 과정을 수행한다. 만일 아직 JEUS 를 설치하지 않았다면 JEUS 설치 안내서를 참조 하기 바란다.

1. JAVA_HOME 이 콘솔에서 설정이 되었는지를 확인한다. java 명령이 실행이 되는지를 확인하기 위해서, 윈도우즈 플랫폼에서 java 실행 파일이 다음과 같은(C:\lang\j2sdk1.4.2\bin\java.exe) 위치에 있다면 JAVA_HOME 은 C:\lang\j2sdk1.4.2 와 같이 설정한다.

   ```
   C:\>set JAVA_HOME=C:\lang\j2sdk1.4.2
   C:\>set JAVA_HOME
   C:\lang\j2sdk1.4.2
   ```

2. jeus 명령을 실행하기 위해서는 JEUS_HOME 이 콘솔에 설정이 되었는지 확인한다. 만일 윈도우즈 플랫폼에서 jeus 실행파일이 다음과 같은(c:\jeus50\bin\jeus.bat) 위치에 있다면, JEUS_HOME 은 C:\jeus50 과 같이 설정 할 수 있다.

   ```
   C:\>set JEUS_HOME=C:\jeus50
   C:\>set JEUS_HOME
   C:\jeus50
   ```

3. CLIENT_HOME , JEUS_WSDIR 와 같은 환경변수가 위와 같은 방법으로 설정이 되었는지 확인한다.

   주의: CLIENT_HOME, JEUS_WSDIR 환경변수는 JEUS JMS 엔진에서는 사용하지 않는다. 하지만 JEUSMain.xml 환경파일 내의 engine-container element 의 다른 엔진이 사용되는 경우에는 위와 같은 환경변수를 사용한다. JEUS Server 안내서에서 JEUS 서버 환경 변수를 참조한다.

4. JEUSMain.xml 환경파일 내의 engine-container element 는 다음의 XML 과 같이 jms 을 포함 해야 한다.

<<*JEUSMain.xml*>>

```
<engine-container>
      ...
     <engine-command>
      <type>jms</type>
      <name>engine1</name>
         </engine-command>
      ...
</engine-container>
```

주의: JEUSMain.xml 은 JEUS_HOME\config\<node name> 디렉토리 아래에 있다. JEUS Server 안내서 내에 JEUS Server 의 디렉토리 구조를 참조한다.

5. JMSMain.xml 은 아래의 태그 내용을 포함하고 JEUS_HOME\config\<node name>\<node name>_jms_<engine name> 디렉토리에 위치 해야만 한다.

<<*JMSMain.xml*>>

```
<?xml version="1.0"?>
<jms-server xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
   <service-config>
      <port>9741</port>
      <blocking-socket>false</blocking-socket>
      <connection-timeout>30</connection-timeout>
   </service-config>
   <durable-subscriber>
        <client-id>durable</client-id>
        <name>durable_test</name>
        <destination-name>ExamplesTopic</destination-name>
   </durable-subscriber>
   <connection-factory>
        <type>topic</type>
        <name>TopicConnectionFactory</name>
        <export-name>TopicConnectionFactory</export-name>
   </connection-factory>
   <destination>
        <type>topic</type>
        <name>ExamplesTopic</name>
        <export-name>ExamplesTopic</export-name>
```

```
   </destination>
</jms-server>
```

6. jeus 명령 스크립트는 JEUS_HOME\bin 디렉토리에서 실행한다.

```
C:\jeus50\bin>jeus.bat
```

7. 터미널/콘솔창을 열고서 jeusadmin 뒤에 컴퓨터 이름을 타이핑 한다 (예: jeusadmin johan). 윈도우에서 hostname 명령 또는 유닉스에서 uname –n 명령은 컴퓨터 이름을 확인 하는데 도움이 된다.

```
C:\jeus50\bin>hostname
johan
C:\jeus50\bin>jeusadmin johan
```

참조 1: CLIENT_HOME, JEUS_WS DIR 환경변수는 JEUS JMS 엔진을 시작하는데 필요하지 않다. 하지만 JEUSMain.xml 환경파일 내의 engine-container element 의 경우에는 다른 엔진 타입을 포함 해야 한다. JEUS Server 안내서 내에 JEUS Manager Properties 를 참조한다.

참조 2: 이 매뉴얼의 나머지 부분에서, JEUS 머신의 이름은 항상 johan 으로 사용한다. 그러므로 johan 을 사용하려는 머신의 이름으로 바꾸기 바란다.

8. 프롬프트에서 JEUS 사용자 이름과 비밀번호를 입력한다.

9. 프롬프트가 나오면, boot 를 입력 하고 enter 키를 누른다. 다음은 예제이다.

```
C:\jeus50\bin>jeusadmin johan
Login name> jeus
Password>
johan>boot
johan _container1
```

10. 잠시 후 프롬프트가 콘솔창에서 다시 출력된다. 이것은 JEUS Server 가 지금 boot 가 되고 명령을 받아 들일 준비가 되어 있다는 것을 의미한다. JMS 엔진은 아래와 같이 명령 allenglist 을 입력 함으로써 boot 가 되었음을 알 수 있다.

```
johan>boot
johan _container1
johan >allenglist
```

```
=======================================
engines in the container johan_container1
johan_jms_engine1
johan_ejb_engine1
johan_servlet_engine1
=======================================
```

11. 시스템 경로에 JEUS_HOME\samples\jms\bin 을 포함하고 있다고 가정하고 터미널/ 콘솔창을 열어 receiver.bat 을 실행한다. 이렇게 함으로써 receiver 은 메시지가 도착 할 때 까지 기다리게 된다.

```
C:\jeus50\samples\manual_examples\JMSGuide\bin>receiver
java –Djava.naming.factory.initial=jeus.jndi.JEUSContex-
tFactory -Djeus.baseport=9736 sample.manual.receiver
wait for messages to arrive..
```

참고: JEUS_HOME, JEUS_BASEPORT 환경변수는 receiver.bat 어플리케이션이 실행되기 전에 설정이 되어야 한다.

참고: JEUS_HOME\samples\manual_examples\JMSGuide\classes 디렉토리에는 receiver.bat 에 필요한 모든 클래스를 포함하고 있다. 만일 정상적으로 시작이 되지 않는다면 필요한 클래스들이 그 디렉토리에 존재하는지를 확인해 본다. 어플리케인션에 대한 소스는 JEUS_HOME\samples\manual_examples\JMSGuide\src 디렉토리에서 확인 할 수 있다.

12. 시스템 경로에 JEUS_HOME\samples\manual_examples\JMSGuide\bin 을 포함하고 있다면 터미널/콘솔창을 열어 sender.bat 을 실행한다.

```
C:\jeus50\samples\manual_examples\JMSGuide\bin>sender
java –Djava.naming.factory.initial=jeus.jndi.JEUSContex-
tFactory -Djeus.baseport=9736 sample.manual.sender
sending a TextMessage ‘Hello World 1’...
sending a TextMessage ‘Hello World 2’...
sending a Non-TextMessage meaning sending ends...
C:\jeus50\samples\manual_examples\JMSGuide\bin>
```

13. receiver 측의 터미널/콘솔창에서 메시지가 도착을 했는지 확인한다. 터미널/콘솔창에서 다음과 같이 보여야 한다.

```
C:\jeus50\samples\manual_examples\JMSGuide\bin>receiver
java –Djava.naming.factory.initial=jeus.jndi.JEUSContex-
```

```
tFactory -Djeus.baseport=9736 sample.manual.receiver
wait for messages to arrive...
receiving a TextMessage 'Hello World 1'
receiving a TextMessage 'Hello World 2'
receiving a Non-TextMessage, so receiving is over.
```

이상으로 JEUS JMS 엔진을 시작하는 방법에 대해서 간략하게 설명했다. receiver 어플리케이션이 메시지를 기다리는 것에 대한 설명과 sender 어플리케인션이 reciver 어플리케이션에게 메시지를 보내는 방법에 대해서 간단하게 알아 보았다.

다음 장은 JEUS JMS 개요에 대해 설명한다.

3 JEUS JMS 의 개요

3.1 소개

JEUS JMS 모듈은 JEUS 시스템의 유용한 컴포넌트이다. 엔터프라이즈 메시징 시스템에 접근하기 위해서 필요한 서브컴포넌트들을 포함하고 있는 엔진이다.

JEUS JMS 는 또한 JEUS Web Application Server 와 함께 완전히 연동이 된다. 예를 들어 Message Driven Bean (JEUS WAS EJB 엔진에서 deploy 됨)은 JEUS JMS 을 이용한다.

다음 절에서 실질적인 JEUS JMS 의 기술적인 세부사항을 보기에 앞서 JEUS JMS 의 개요에 대해서 다룬다. 개요에 대한 설명은 다음과 같다.

- JEUS JMS 특징
- JEUS JMS 컴포넌트
- JEUS JMS 아키텍처
- JMS Server
- JMS Server 의 하위 컴포넌트
- JMS 클라이언트 어플리케이션
- JMS 디렉토리 구조
- JEUS JMS 관리 툴

주의: 이 장은(본 메뉴얼의 모든 장에서) J2EE 또는 JMS 1.1 스펙의 기본적인 내용을 다루지는 않는다. 이런 기술적인 내용에 대한 지식들은 알고 있다고 가정한다. 이 메뉴얼의 내용을 이해하기 전에 관련 내용에 대한 좋은 참고서적이나 스펙을 참조하기 바란다.

3.2 JEUS JMS 의 특징

JEUS JMS 는 엔터프라이즈 메시지 미들웨어 뿐만 아니라 J2EE 컴포넌트의 특징을 포함하고 있다. 주목할 특성은 다음과 같다.

- JMS 1.1 스펙을 완전히 지원한다.
- fail over 을 지원한다. 이것은 서버측에서 예상하지 않은 장애가 발생할 경우에도 메시지를 받을려고 기다리는 모든 consumer 들에게 메시지를 보내는 것을 보장한다.
- 다른 JMS Server 그리고 다른 운영체제에서도 메시지 처리가 가능하다.
- 사용자 어플리케이션의 세팅에 따라서 또는 자동적인 트랜잭션의 요구에 참여 하므로써 global transaction 이나 local transaction 에 참여 한다.
- 사용자 어플리케이션과 서버 destination 사이에 최적화된 메시지를 보낼 수 있도록 한다.
- 사용자들에게 실행중인 client API 뿐만 아니라 jmsadmin 을 통해서 JMS Server 를 관리 할 수 있도록 한다 (부록 A 참조).

3.3 JEUS JMS 컴포넌트

이 절에서는 JEUS JMS 가 자신이 포함하고 있는 컴포넌트 또는 함께 상호작용을 요구하는 기본적이고 중요한 컴포넌트들을 설명한다.

메인 컴포넌트

1. JEUS JMS engine 은 Administration 객체의 서버 컴포넌트들을 포함하고 있고 JEUS Server 의 엔진 컨테이너내에 위치한다.

   주의: 세부적인 엔진 컨테이너 구조와 JEUS Server 을 참조하기 위해서는 JEUS Server 안내서를 참조하기 바란다. JEUS JMS Server 와 JEUS JMS engine 는 같은 용어로 사용한다.

2. **JNDI** (Java Naming and Directory Interface) 는 bind 또는 lookup 기능을 제공한다.

3. **Client application** 은 메시지를 보내고 받는 역할을 한다.

4. **Persistent Storage** 은 persistent 메시지 데이터와 트랜잭션 데이터를 저장한다.

서브 컴포넌트

1. **ConnectionFactory** 는 클라이언트 어플리케이션이 JEUS JMS 엔진과 연결을 위해서 사용이 된다.

2. **Topic 는** publish-subscribe 메시지 모델을 지원하는 destination 이다.

3. **Queue** 는 point-to-point 메시지 모델을 지원하는 destination 이다.

JEUS JMS Server 는 서버 컴포넌트로서 관리 객체들을 가지고 있기 때문에 JNDI 서비스로 bind 할 수 있다. 그래서 JMS 클라이언트 어플리케이션들은 일반적인 JNDI lookup 기능을 통해서 관리 객체들을 얻을 수 있다. ConnectionFactory 를 가진 클라이언트들은 JEUS JMS Server 로 연결할 수 있고 Topic 또는 Queue 로 연결하여 JEUS Server 측의 destination 으로 메시지를 보낼 수 있다.

Client Application 에서 persistent 로 기술하여 메시지를 보내면, JEUS JMS Server 의 Persistent Storage 로 저장이 된다. Persistent Storage 는 파일이나 데이터베이스에 저장한다. 하지만 일반적으로 JEUS JMS 는 데이터베이스를 이용한다.

이 장에서는 JEUS JMS 컴포넌트들에 대해서 간략하게 설명하고 있을 뿐만 아니라 엔터프라이즈 메시지 서비스를 지원하는 방법에 대해서 설명하고 있다.

3.4 JEUS JMS 의 아키텍처

다음 그림은 이전 절에서 언급이 된 JEUS JMS 아키텍처를 설명한다. [그림 1].

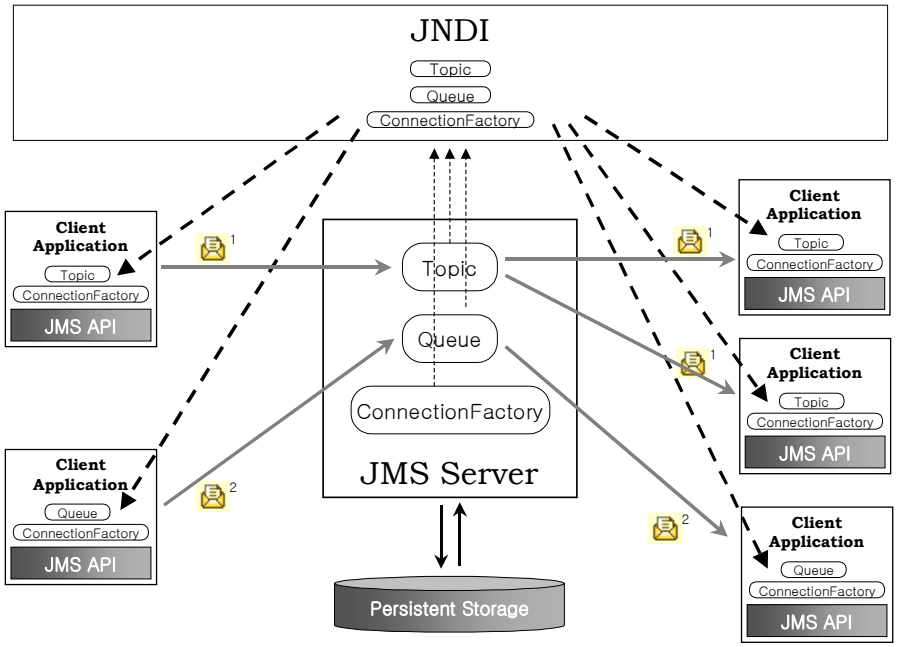

그림 1. JEUS JMS 의 아키텍처

3.5 JMS Server

JMS Server 는 메시지 서비스를 지원하는 최상위 컴포넌트이다. 하나의 JMS 엔진이 하나의 JMS Server 인스턴스와 연결된다.

서버 인스턴스는 클라이언트 요구에 대해 특정한 port 를 listen 하는 JEUS 엔진 컨테이너 내에 하나의 엔진으로 운영된다. listen 하는 port 는 JMSMain.xml 환경파일에 설정된다. 만일 설정이 되어 있지 않다면 JMS 엔진이 부트가 되지 않는다. 만약 하나 이상의 서버 인스턴스들이 동일한 머신에서 부트가 된다면 같은 port 를 사용하고 있는지 확인해야 한다.

서버 인스턴스는 topic 또는 queue 형태의 destination 과, topic 또는 queue 에 해당하는 connection factory 를 가진다. 각각은 JMS Server 인스턴스가 시작하기 전에 정적으로 환경을 설정할 수 있고, 또 실행 중에 동적으로 증가 또는 감소 시킬 수도 있다.

Persistent storage 는 예상하지 않은 서버의 장애가 발생할 때 메시지 전달을 보장하기 위해서 구성되며 파일 시스템이나 데이터베이스가 될 수 있다. 그러나 클러스터링 환경일 때, 동일한 storage 설정을 다른 JMS Server 인스턴스들이 공유하여 사용할 수는 없다

참고: JEUS JMS 의 현재 버전에서는 데이터베이스와 파일을 persistent storage 로 지원한다.

3.6 JMS Server 의 하위 컴포넌트

JEUS JMS Server 는 하위 컴포너트로서 thread pool, destination, durable subscription 그리고 connection factory 을 구성하고 있다. 이런 컴포넌트들은 JMS 클라이언트 어플리케이션들과 JEUS administrator 에서도 볼 수 있다.

JEUS JMS Server 는 클라이언트의 요구를 처리하기 위해서 Worker Thread 를 포함하는 하나의 thread pool 을 가지고 있다. 이런 속성들은 환경 설정에서 세팅된다.

logger 는 엔진과 관련된 활동을 저장하거나 보여 준다. 즉 클라이언트 접근 정보나 클라이언트 메시지 처리에 대한 정보를 볼 수 있다.

destination 는 서버측에서 메시지가 저장되는 저장소로써 볼 수 있다. JMS 스펙에서 언급된 destination 은 queue 또는 topic 이 있고 클라이언트 어플리케이션은 JNDI 를 통해서 이것을 얻을수 있다. topic 은 Pub/Sub 메시징 모델을 지원하고 queue 는 Peer-to-Peer 메시징 모델을 지원한다.

connection factory 는 클라이언트 애플리케이션에서 connection 을 만드는데 사용한다. 서버측에서 만들어지고 구성된다. 이것은 또한 클라이언트 어플리케이션에 의해서 JNDI 을 통해서 찾을 수 있다.

fail over / backup 기능을 위한 연속적인 스토리지가 있다. 이 연속적인 스토리지를 위해서 database 와 file 시스템을 사용한다.

3.7 JMS Client Application

기본적으로 JMS 클라이언트 어플리케이션은 메시지를 보내거나 또는 동기적으로 혹은 비동기적으로 메시지를 받는다. JNDI lookup 기능을 통해서 connection factory 와 destination 을 찾을 수 있다. 주어진 connection factory 와 destination 을 얻고나서 JMS 1.1 스펙의 기준이 되는 표준 JMS API 을 이용해 실행 한다.

또한 클라이언트 어플리케이션은 JNDI 을 통해서 얻은 connection factory 에 thread pool 을 가지고 있다. 이것은 클라이언트 어플리케이션이 JEUS JMS server 에게 요구한 것을 처리하는데 사용된다.

3.8 JMS 디렉토리 구조

이 절에서 JEUS JMS 을 실행할때 알아야 하는 JEUS 의 기본적인 디렉토리 구조에 대해 설명하고 있다[그림 2].

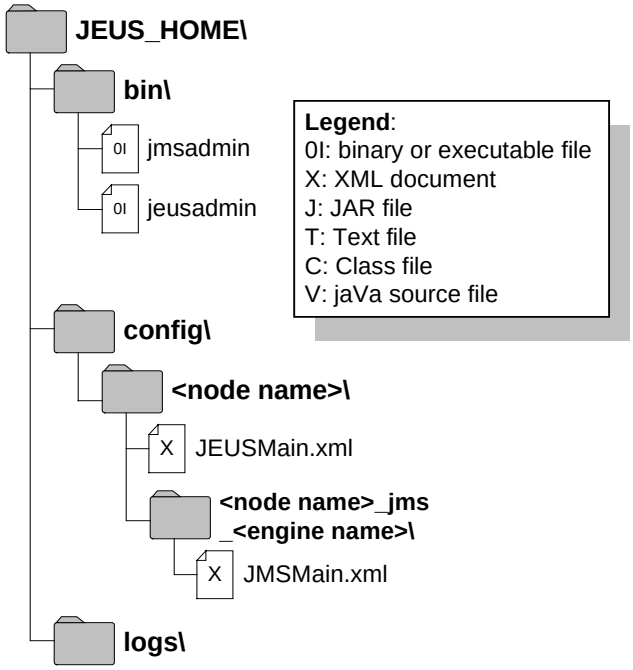

그림 2. JMS 디렉토리 구조

중요 디렉토리는 다음과 같다.

- JEUS 제품을 인스톨 할때 선택하는 디렉토리는 JEUS_HOME 디렉토리이다(예를 들어 c:\jeus50).

- jmsadmin console tool 을 포함하는 디렉토리는 JEUS_HOME\bin 디렉토리이다.(부록 A 참조).

- JEUS_HOME\config 디렉토리는 JEUS 에서 환경구성 파일들을 노드별로 관리한다. config 디렉토리 아래에는 노드이름과 같은 디렉토리가 있는데 여기에 해당 노드의 모든 엔진들의 환경구성 파일들이 위치한다. 각 엔진들의 환경구성 파일은 엔진들의 이름으로 된 디렉토리 아래에 놓인다. 그리고 JMS Server 의 환경설정 파일인 JMSMain.xml 는 JEUS_HOME\config\<node name>\<node name>_jms_<eninge name> 디렉토리에 위치한다. JMSMain.xml 는 JMS Server 에 관한 모든 설정들을 담고 있다. 더 많은 정보를 위해서는 JEUS Server 안내서를 참조한다.

- logs 디렉토리는 JMS 엔진에 대한 로그 정보를 저장한다.

참조 : 전체적인 JEUS 구조와 세부적인 설명과 관련된 내용은 JEUS Server 안내서를 참조한다.

3.9 JEUS JMS 관리 툴들

JEUS JMS 을 통제하기 위한 3 가지 툴이 있다. 일반적인 엔진과 관련된 명령은 jeusadmin, 웹 관리자 그리고 jmsadmin 을 통해서 통제 가능하다.

웹 관리자는 JEUS JMS 엔진을 생성하고, 생성된 엔진의 설정을 세팅할 수 있다. 또한 엔진을 제어하고 엔진의 상태를 체크하는 등의 관리 기능도 수행할 수 있다.

모든 툴의 script 가 위치하는디렉토리는 JEUS_HOME\bin 디렉토리이다.

3.10결론

JEUS JMS Server 에 대한 고수준의 구조적인 내용에 대해 설명하였다.

JEUS JMS 엔진이 전반적으로 JEUS 시스템과 어떻게 연관이 되는지를 살펴보았으며, 중요한 서브 컴포넌트로는 무엇이 있고, JEUS JMS 가 JMS 1.1 엔터프라이즈 메시징 시스템과 어떻게 작용을 하는지 살펴 보았다.

JMS Server 로 message 들을 보내고 그것으로부터 메시지를 받는 예제를 살펴 보았다. JMS 어플리케이션에 대한 중요한 컴포넌트들에 대한 간략한 설명을 하였다.

다음 장에서 JEUS JMS Server 에 대한 환경 설정을 자세히 살펴본다.

4 JMS Server 구성

4.1 소개

JEUS System 의 JEUS JMS Server 는 클라이언트 어플리케이션에 기준이 되는 JMS 스펙 1.1 을 구현한 엔트프라이즈 메시징 시스템을 제공한다.

JEUS JMS Server 는 destination, connection, 그리고 persistent storage 에 해당하는 데이터베이스와 같은 관리 객체를 포함하고 있다. 예를 들어, JEUS JMS Server 는 JNDI bind 를 통해서 destination 을 얻고 클라이언트 어플리케이션에 의해서 얻어진 destination 에 메시지를 저장하는 Worker Thread 를 가지고 있다.

이런 개념과 환경 구성에 대한 모든 것들을 이번 장에서 설명한다. 그리고 JMS 어플리케이션에 대한 모든 관리 기능들에 대해서도 다룬다.

JMS Server 에 대한 모든 환경구성은 JEUS_HOME\config\<node name>\<node name>_jms_<engine name>\ 디렉토리에 있는 JMSMain.xml 파일에서 만들어진다. 이 파일은 하나의 최상위 레벨의 XML element 인, <jms-server> 태그로부터 시작해서, 여러 하위 태그들로 구성되어 있다.

예제:

<<*JMSMain.xml*>>

```
<?xml version="1.0"?>

<jms-server xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    ...
</jms-server>
```

4.2 Destination 구성

Destination 은 JMS 클라이언트 어플리케이션들이 message 들을 보내고 받는 장소이다. 각 destination 은 JMSMain.xml 파일의 `<destination>` 태그로 구성이 된다.

- **type:** destination 의 type 을 명확하게 한다. 이것은 queue 또는 topic 으로 설정한다.
- **name**: desination 의 이름을 명시한다.
- **export-name**: JNDI 을 통해서 bind 되는 이름을 넣는다.
- **multiple-**receiver: destination 이 queue 인 경우 여러 MessageConsumer 를 허용할 것인가 아닌가를 결정한다. Default 값은 false 로 허용하지 않을 경우이다. 이 경우 queue 로 오는 메시지는 queue 의 receiver 중 하나의 receiver 에만 전달된다.
- **relay**: 이 destination 으로 오는 메세지를 다른 JEUS JMS Server 의 destination 으로 보내고자 할 때 사용한다. 이 설정은 JEUS JMS Server 들의 clustering 을 위해 사용할 수 있고 자세한 설정 사항에 대해서는 부록 B 의 XSD element 설정을 참고하기 바란다.

export-name element 는 부가적인 것이다. 만일 설정하지 않으면, name 과 같다. 다음은 destination 을 구성하는 예제이다.

<<*JMSMain.xml*>>

```
<jms-server>
    ...
    <destination>
        <type>queue</type>
        <name>myDestName</name>
        <export-name>myDestJndi</export-name>
        <multiple-receiver>false</multiple-receiver>
    </destination>
    ...
</jms-server>
```

4.3 Durable Subscriber 구성

DurableSubscriber 는 JMS 클라이언트 어플리케이션 들이 메시지를 보내고 받는 역할을 하는 Topic 에 대해서만 구성이 된다. 이것은 JMSMain.xml 에 세팅되어서 JMSServer 가 시작될 때 등록된다.

각각의 durable subscriber 는 `<durable-subscriber>` element 로 구성이 된다. 기본적인 설정 태그는 다음과 같다.

- **client-id:** 클라이언트의 유일한 값을 설정하기 위한 것이다. connection factory 내에서 뿐만 아니라 durable subscriber 내에서 모든 client-id 값은 중복 되어서는 안된다.

- **name:** durable subscription 이름을 설정한다.

- **destination-name:** 이 durable subscriber 의 목적지가 되는 topic destination 을 설정하기 위한 것이다. 같은 이름을 가지는 destination 이 JMS Main.xml 에 포함되어 있어야 한다.

- **message-selector:** 이런 durable subscriber 을 위한 message 들를 여가하기 위해서 설정한다.

다음의 XML 일부는 destination 을 구성하기 위한 예제이다.

<<*JMSMain.xml*>>

```
<jms-server>
    ...
    <durable-subscriber>
        <client-id>sports_client</client-id>
        <name>durable_sports</name>
        <destination-name>SportsTopic</destination-name>
        <message-selector>
           JMSType = 'sports' and age = 20
        </message-selector>
    </durable-subscriber>
    ...
</jms-server>
```

4.4 ConnectionFactory 구성

Connection factory 는 클라이언트 어플리케이션 사용을 편리하게 하기 위해서 JEUS JMS Server 에서 connection 을 만들고 저장하는데 사용이 된다. 각 connection factory 는 JMSMain.xml 안에서 `<connection-factory>` 태그로 구성된다. 기본적으로 구성하는 태그는 다음과 같다.

- **type:** connection factory 의 타입을 정의하기 위한 설정이다. queue 나 topic 으로 구성된다.

- **name**: connection factory 의 이름을 설정한다.

- **export-name**: JNDI 에 bind 된 이름을 설정한다.
- **client-id**: topic 형태의 connection factory 에 대해서 default client-id 값을 설정한다. client-id 값은 connection factory 들간에 중복 되어서는 안된다.
- **Thread** - pool: 이 ConnectionFactory 로부터 생성던 Connection 의 thread pool 의 속성을 정한다.
- **check-security** : 이 ConnectionFactory 에 대해 connection 을 생성할 때 security check 를 할것인지의 여부를 지정한다. Security 설정에 대해서는 JEUS Security 매뉴얼을 참고하기 바란다.

만일 export-name element 가 생략이 된다면, <name> element 의 값으로 세팅된다.

client-id element 는 connection factory 형태가 topic 일 때만 의미가 있다.

client 측면에 있어서 connection factory 는 JMS Server 와 통신하는데 사용 하는 connection 을 만드는 하나의 thread pool 을 가진다. 기본적으로 클라이언트 측에서 thread pool 은 최대 쓰레드 수 보다 더 많이 생성할 수는 있지만, 최대 쓰레스 수 이외의 쓰레스는 보관되지 않는다. 이에 대한 구성 방법은 Thread pool 의 구성 장을 참조한다.

다음은 connection factory 을 구성하는 예제이다.

<<JMSMain.xml>>

```
<jms-server>
    ...
    <connection-factory>
        <type>queue</type>
        <name>ConFacSports</name>
        <export-name>ConFacSportsJndi</export-name>
        <client-id>client_sports</client-id>
    </connection-factory>
    ...
<jms-server>
```

4.5 Thread Pool 구성

JEUS JMS Server 내의 thread pool 는 클라이언트가 요구하는 connection 을 만들고, 메시지를 보내고 메시지를 받는 모든 처리를 위해서 사용하는 Worker Thread 를 포함하고 있다.

JMSMain.xml 파일 내에, `<thread-pool>` 태그는 JMS server 가 사용하는 thread pool 에 대한 속성을 지정한다. 이 element 에 대한 자세한 설명은 부록 B 를 참고하기 바란다.

4.6 Logger 구성

JEUS JMS 는 두 종류의 로그 정보를 제공한다. 하나는 JMS Server 처리에 대한 정보를 저장하는 error-log 로 JEUSMain.xml 에서 <engine-command> 에서 지정한다. 다른 하나는 클라이언트의 connection 과 disconnection 에 대한 정보를 기록하는 access-log 로 JMSMain.xml 파일 내의 `<jms-server>` element 아래의 태그에 위치한다. access-log 나 error-log 나 설정하는 방법은 JEUS 의 다른 logger 설정과 같으므로 JEUS Server 매뉴얼의 logging 항목을 참고하기 바란다.

4.7 Persistent 스토리지의 구성

JMS 서버가 다운이 될 경우에, 백업 이나 failover 의 기능을 제공하기 위해서, JMS 환경파일에 퍼시스턴드 스토리지 관련 부분을 구성해야 한다. 파일이나 database 둘중에 하나를 JMS 환경 파일에 지정해야 한다.

데이터 베이스 벤더로 oracle, mssql, informix, cloudscape, db2, sybase 등을 지원한다. 만일 JEUSMain.xml 파일내의 벤더 이름이 database 엘리먼트내의 unknown 또는 others 이면, JEUS JMS 는 자동적으로 벤더 타입을 결정 짓는다. 이런 경우에는, 다음의 옵션은 특정한 database 벤더 타입을 결정을 하는데 사용이 된다(jeus 스크립트내의 속성값을 세팅).

-Djeus.jms.server.blackboxDB=<database vendor name>

여기에 가능한 값은 oracle, mssql, informix, cloudscape, db2, sybase 이다.

파일을 기본으로 하는 스토리지를 구성하기 위해서, JMSMain.xml 파일내의 <storage><file-storage> 엘리먼트를 추가 해야 한다.

이 각각의 element 에 대한 설명은 부록 B 의 XML Schema reference 를 참고하기 바란다.

4.8 Clustering 구성

JMS Servers 의 클러스터링은 <destination> element 의 relay 설정으로 구성된다. 이를 통해 얻을수 있는 효과는 부하분산과 failover 이다. 즉, 어떤 Queue 에 대해 다른 JMS Server 의 Queue 로 relay 를 걸어두고 이 Queue 의 메시지를 처리할 client 를 반씩 나누어 어느 하나의 Server 에 접속해 놓는다면 JMS Server 에 가해지는 부하가 반으로 줄어들 것이다. 또한 이런 경우 하나의 Server 가 down 된다면 나머지 한 Server 가 message 처리를 계속해줄수 있을 것이다. 이러한 relay 의 설정은 부록 B 의 XML Schema reference 를 참고하기 바란다.

4.9 결론

이 장에서 JEUS server 내에 JMS 엔진을 구성하는 방법에 대해 살펴 보았다.

destination, durable subscriber, connection factory, database, thread pool, logger 의 설정에 대한 필요성을 언급했다.

다음 장에서 JMS engine 을 관리하는 방법에 대해서 언급한다.

5 JMS Server 관리

5.1 소개

본 장에서는 JEUS JMS Server 를 관리하는 방법에 대해서 언급하고 엔진의 시작과 정지 그리고 connection factory 등을 생성하는 내용을 포함하고 있다. jeusadmin, jmsadmin 툴을 가지고 여러 명령을 실행해 본다.

5.2 jeusadmin 을 사용한 관리

이 절에서 JEUS JMS 부트와 다운 시키는 방법에 대해서 설명한다.

jeusadmin 을 통해서, JEUS JMS engine 을 부트시키고 다운 시키는 것이 가능하다.

JEUS JMS engine 을 부트 시키기 위해서, starteng <nodename>_jms_<engine X> 명령을 사용한다. 여기에서 <nodename>는 JEUS 가 운영중인 머신의 이름을 말한다. 터미널/콘솔창에서 Windows 경우에는 hostname 그리고 UNIX 의 경우에는 uname –n 을 사용해서 확인 할 수 있다. 마지막에 붙는 engineX 는 JEUSMain.xml 파일내에 `<engine-command>` 하위 태그인 `<name>` element 에서 설정한 값과 같은 JMS 엔진의 이름을 설정한다. 만일 JEUSMain.xml 파일에 다음과 같은 XML 일부를 가지고 있다면 그것 또한 JMSMain.xml 파일이 JEUS_HOME\config\johan\johan_jms_engine1 내에 위치함을 확인 할 수 있다. 또한, boot 명령은 JEUSMain.xml 파일내 등록된 모든 엔진을 시작시킬 수 있다.

<<JEUSMain.xml>>

```
...
<engine-container>
      ...
     <engine-command>
      <type>jms</type>
      <name>engine1</name>
  </engine-command>
    ...
```

```
</engine-container>
```

참고: jeusadmin 은 JEUS_HOME\bin 디렉토리에 있다. JEUS Server 에 대해서는 JEUS Server 안내서를 참조 하거나 이 안내서에 있는 2.2 따라하기를 참조한다.

JEUS JMS engine 을 다운 시키기 위해서는 downeng <nodename>_jms_<engineX> 명령을 사용한다. down 명령은 JMS 엔진을 포함하는 모든 엔진을 중지시킨다.

5.3 jmsadmin 을 사용한 관리

이 절에서는 JEUS JMS 모니터링하는 방법에 대해 간략하게 설명한다.

jmsadmin 툴을 통해서, JEUS JMS engine 이 부트된 후에 JEUS JMS Server 를 모니터링 할 수 있다. 이런 jmsadmin 은 터미널/콘솔 창에서 보안체크를 통해 확인하고 JEUS JMS 엔진과 연결할 수 있다. jmsadmin 툴은 JEUS_HOME\bin 디렉토리에 위치 한다. 부록 A 에서 jmsadmin 툴의 세부적인 사용을 참조한다.

다음은 JEUS JMS 관리와 연관된 jmsadmin 의 명령들이다.

- **dest <destName> | -m <destName> [<message selector>] | –c (–q| -t) <destName> <jndiName> | -r <destName> :** destination 에 관련된 명령이다. destination name 만 지정하면 해당 destination 에 대한 정보가 나오고 –m 을 지정할 경우에는 그 destination 에 저장된 메시지의 정보를 볼수 있다. 이 경우 message selector 를 지정해서 특정 메시지만 확인할수도 있다. –c 를 사용할 경우에는 destination 을 생성하기 위함이며 –q 의 경우 queue, -t 의 경우 topic 을 생성할수 있다. 그리고 –r 을 사용할 경우에는 해당 destination 을 제거할 수 있다.

- **conf <factoryName> | –c –q <factoryName> <jndiName> [<max-thread>] | –c –t <factoryName> <jndiName> [<max-thread> [<client-id>]] | -r <factoryName>** : connection factory 에 관련된 명령이다. Factory name 만 제공하면 그 connection factory 의 정보를 볼 수 있다. –c 를 사용하면 connection factory 를 생성할 수 있고 이때 –q 는 queue connection factory, -t 는 topic connection factory 이다. –r 을 사용하면 지정된 connection factory 를 제거할 수 있다.

주의 : 이것은 JEUS JMS 엔진을 시작 시키고 멈추게 할 수는 없다. JEUS 가 부트가 된 후에 활성화 될수 있다.

5.4 결론

이 절에서 jeusadmin 을 사용해서 JEUS JMS 의 시작과 멈추는 방법에 대해서 설명했다. JEUS JMS 가 시작이 될 때, jmsadmin 과 JMSCommander 를 통해서, administrator 와 JMS 클라이언트 어플리케이션은 connection factory 와 destinations 을 추가, 제거할 수 있도록 한다.

다음 장에서는 JEUS JMS 모니터링하는 방법에 대해서 설명한다.

6 JMS Server 모니터링

6.1 소개

이 절에서 JEUS JMS Server 의 모니터링에 대해서 설명하고 있다. jeusadmin, jmsadmin 이런 두 가지 툴을 가지고 모니터링하는 방법에 대해서 설명한다

6.2 jeusadmin 을 사용한 모니터링

jeusadmin 을 사용하므로써 단지 타깃인 jms 엔진이 boot 되었는지 down 되었는지 확인 가능한다. jeusadmin 은 allenglist 명령을 통해서 실행된다. 다음 터미널/콘솔 툴의 예제를 보여주고 있다.

```
johan>
johan>allenglist
johan_ejb_engine1
johan_jms_engine1
johan>
```

6.3 jmsadmin 사용한 모니터링

이 장에서 JEUS JMS 모니터링하는 방법에 대해서 간략하게 설명하고 있다.

jmsadmin 을 통해서, JEUS JMS engine 이 부트가 된 후에 JEUS JMS Server 를 모니터링 할수 있다. 터미널/콘솔 창은 security 체크를 통해서 JEUS JMS engine 과 연결할 수 있다. jmsadmin 명령 스립트는 JEUS_HOME\bin 디렉토리에 위치한다. 부록 A 에서 jmsadmin 의 세부적인 사용방법을 참조한다.

다음은 JEUS JMS 관리와 관련된 jmsadmin 명령에 대해서 설명한다.

- **serverall** : 완전한 server 에 관련된 모든 정보를 얻을 수 있다.
- **server** : 일반적인 server 정보를 얻을 수 있다.
- **dest <DestinationName>** : 해당하는 이름의 destination 정보를 얻을 수 있다.

- **con <ConnectionFactory Name>** : 해당하는 이름의 connection factory 정보를 얻을 수 있다.
- **client <Client Number>** : 해당 client 정보를 얻을 수 있다.
- **durable <TopicName> <ClientID> <Name>** : durable subscription 정보를 얻을 수 있다.

참고: 부록 A 의 예제를 참조한다. jmsadmin 으로는 JMS 엔진을 시작시키고 멈추게 할 수는 없다. 단지 JEUS JMS 엔진이 시작된 후에 이용할 수 있다.

6.4 결론

이 절에서 jeusadmin, jmsadmin 를 사용해서 JEUS JMS 엔진을 모니터링하는 방법에 대해 설명했다. jmsadmin, administrator 를 통해서 여러가지 실행중인 정보를 볼수 있었다.

다음 장에서 JEUS JMS 를 이용해서 프로그램 하는 방법에 대해 설명할 것이다.

7 JEUS JMS 프로그래밍

7.1 소개

다음 절에서는 클라이언트 어플리케이션이 JEUS JMS 를 어떻게 개발하고 작성하는지에 대해서 설명한다. 이 매뉴얼은 J2EE JMS 1.1 스펙에 대한 배경 지식을 가지고 있다고 가정하고 JEUS JMS 에 적합한 프로그래밍 모델과 개발 방법에 초점을 맞추고 있다.

참고: 위에서 언급된 JMS API 대한 더 많은 설명을 위해서, J2EE JMS 1.1 스펙 또는 튜토리얼을 참조하기 바란다.

7.2 일반적인 어플리케이션 구조

이 절에서는, JEUS JMS 클라이언트 어플리케이션의 일반적인 구조를 보여준다. 그리고 어플리케이션의 구조, 그 자체의 필요성에 따라서 다양화 될 수 있다는 것을 보여준다.

JMS 클라이언트 어플리케이션은 다음 단계에 따라 수행된다.

- Step 1: ConnectionFactory lookup 한다.
- Step 2: Connection 을 만든다.
- Step 3: Session 을 만든다.
- Step 4: Destination lookup 한다.
- Step 5 : Message Producer 또는 Message Consumer 를 만들거나 Message Listener 를 등록한다.
- Step 6: Connection 을 시작한다.
- Step 7: 어떤 특정한 작업의 어플리케이션을 실행한다.
- Step 8 : Connection 을 닫는다.

1 에서 4 까지의 단계는 JMS 클라이언트 어플리케이션에 대해 리소스를 설정하는 단계로 볼 수 있다. 어플리케이션의 작업 전에 리소스 설정을 해야한다. 단계 7 에서 실행한 후에 connection 을 닫는 것과 같은 리소스의 해제는 반드시 해줘야한다.

단계 4 는 어플리케이션에서 다르게 처리할 수 있다. 예를 들어 Destination 을 lookup 하거나 session.createDestination 을 통해서 만들 수 있다. 다음 절에서 이에 대한 설명을 한다.

기본적인 어플리케이션 구조는 이 절에서 설명했다. 그리고 JMS 어플리케이션의 일반적인 두가지 형태는 다음 두 절에서 연속해서 설명한다.

7.3 Messages 보내기

7.3.1 소개

Destination 에 messages 를 보내는 일반적인 JMS 어플리케이션을 이전 절에서 살펴 보았고, 단계별로 연속해서 이 절에서 소개한다. Queue Destination 과 Topic Destination, 두 가지의 경우는 이 장에서 다룬다.

7.3.2 Queue 로 메시지를 보내기 위한 리소스의 설정

우선, Queue 에 메시지를 보내는 클래스의 이름을 qSender 라고 하겠다. 그리고 qSender 클래스는 다음과 같이 필요한 package 들을 import 한다.

<<qSender.java>>

```
package sample.mannual;
import javax.jms.*;
import javax.naming.*;

public class qSender {
    Context ctx = null;
    QueueConnectionFactory qConFactory = null;
    QueueConnection qCon = null;
    QueueSession qSes = null;
    Queue queue = null;
    QueueSender qSender = null;
      ...
```

단계 1, “ConnectionFactory 의 lookup” 뿐만 아니라 단계 2, 단계 3, 단계 4, 단계 5 에서도 initResource() 메소드를 추가한다. 이 메소드는 리소스를 사용하기 전에 초기화시켜준다.

```
public void initResource() {
    try {
        ctx = new InitialContext();
    } catch (NamingException ex) {
        System.err.print("Cannot create Context :" +
              ex.toString() );
          System.exit(1);
      }

      //Step 1: Lookup a ConnectionFactory
      try {
          qConFactory = (QueueConnectionFactory)
              ctx.lookup("QueueConnectionFactory");
      } catch (NamingException ex) {
          System.err.print("Cannot find
              QueueConnectionFactory in JNDI :" +
              ex.toString());
          System.exit(1);
      }

      //Step 2: Create a Connection
      try {
          qCon = qConFactory.createQueueConnection();
      } catch ( JMSException ex) {
          System.err.println("Cannot create
          QueueConnection :" + ex.toString());
          System.exit(1);
      }
      //Step 3: Create a Session
      try {
          qSes = qCon.createQueueSession(
              false,Session.AUTO_ACKNOWLEDGE );
      } catch (JMSException ex) {
          System.err.println("Cannot create
          QueueSession :" + ex.toString());
      }
```

```
      //Step 4: Lookup a Destination
      try {
          queue = (Queue)ctx.lookup("ExamplesQueue");
      } catch (NamingException ex) {
          System.err.println("Cannot find ExamplesQueue
          in JNDI :" + ex.toString());
      }
      //Step 5: Create a  MessageProducer or a MessageConsumer,
           or register a MessageListener
      try {
          qSender = qSes.createSender(queue);
      } catch( JMSException ex) {
          System.err.println("Cannot create
          QueueSender :" + ex.toString());
          System.exit(1);
      }
  }
```

이 예제는, 리소스를 초기화가 실패하면, 어플리케이션은 실행을 멈추게 된다. 그러나 실제 어플리케이션은 exception 을 발생 시키고 장애 사항을 관리하게 된다.

JEUS Server 에서 JNDI initial context 를 세팅하기 위해서는 JNDI 프로퍼티가 필요하다. 위의 예제에서는 다음 JVM 파라미터를 사용한다.

- **-Djava.naming.factory.initial= jeus.jndi. JEUSContextFactory**
- **-Djava.naming.provider.url=<JEUS Server IP address>**

이런 JNDI 프로퍼티는 다음과 같이 JEUS JMS 클라이언트 어플리케이션 내에 세팅할 수 있다.

<<qSender.java>>

```
public class qSender {
    public final static String JNDI_FACTORY =
        "jeus.jndi.JEUSContextFactory";
    public final static String PROVIDER_URL =
        "localhost";
    . . .
    public void initResource() {
        try {
```

```
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                JNDI_FACTORY);
            env.put(Context.PROVIDER_URL,
                PROVIDER_URL);
            ctx = new InitialContext(env);
        } catch (NamingException ex) {
            System.err.print("Cannot create
                Context :" + ex.toString() );
            System.exit(1);
        }
    . . .
```

참고 : 세부적으로 JNDI 를 다루려면, JEUS Server 안내서를 참조한다. 만일 JMS 어플리케이션이 Servlet 또는 EJB 에서 실행 중이라면 JNDI 파라미터 세팅은 필요하지 않다.

Connection 은 다른 쓰레드 중에서 공유 될 수 있다. 하지만 session 은 공유 될 수 없다. session 을 사용 할 수 있는 쓰레드는 session 을 만든 쓰레드만이 가능하다. 2 개 이상의 쓰레드가 session 을 공유하려 할때는 session 을 보장하지 않는다.

타깃 destination 의 레퍼런스을 얻기위한 것으로는 여러 방법이 있다. destination 을 만드는 것과 관련하여 4.2 절과 7.9 절을 참조한다.

7.3.3 Topic 으로 메시지를 보내기 위한 리소스 세팅

이 경우에 대해, 클리스 이름은 tPublisher 로 한다.

```
package sample.mannual;
import javax.jms.*;
import javax.naming.*;
```

<<tPublisher.java>>

```
public class tPublisher {
    Context ctx = null;
    TopicConnectionFactory tConFactory = null;
    TopicConnection tCon = null;
    TopicSession tSes = null;
    Topic topic = null;
    TopicPublisher tpublisher = null;
```

```
    ...
```

단계 1, “ConnectionFactory 의 lookup” 뿐만 아니라 단계 2, 단계 3, 단계 4, 단계 5 에서도 initResource() 메소드를 추가한다. 이 메소드는 리소스를 사용하기 전에 초기화시켜준다.

```
public void initResource() {
    try {
        ctx = new InitialContext();        }
    catch (NamingException ex) {
        System.err.print("Cannot create Context :" +
            ex.toString() );
        System.exit(1);
    }
    //Step 1: Lookup a ConnectionFactory
    try {
        tConFactory = (TopicConnectionFactory)
            ctx.lookup("TopicConnectionFactory");
    } catch (NamingException ex) {
        System.err.print("Cannot find
            TopicConnectionFactory in JNDI :" +
            ex.toString());
        System.exit(1);
    }

    //Step 2: Create a Connection
    try {
        tCon = tConFactory.createTopicConnection();
    } catch ( JMSException ex) {
        System.err.println("Cannot create
            TopicConnection :" + ex.toString());
        System.exit(1);
    }
    //Step 3: Create a Session
    try {
        tSes = tCon.createTopicSession(
            false,Session.AUTO_ACKNOWLEDGE );
    } catch (JMSException ex) {
        System.err.println("Cannot create
            TopicSession :" + ex.toString());
```

```
      }
      //Step 4: Lookup a Destination
      try {
          topic = (Topic)ctx.lookup("ExamplesTopic");
      } catch (NamingException ex) {
          System.err.println("Cannot find ExamplesTopic
              in JNDI :" + ex.toString());
      }

  //Step 5: Create a  MessageProducer or a MessageConsumer, or
register a MessageListener
      try {
          tpublisher = tSes.createPublisher(topic);
      } catch( JMSException ex) {
          System.err.println("Cannot create
              TopicPublisher :" + ex.toString());
          System.exit(1);
      }
    }
```

7.3.4 Connection 시작

Connection 은 initResource 메소드에서 초기화가 되고, 그런 후에 startConnection 메소드에서 시작된다. Topic 의 경우도 동일하게 적용된다.

```
public void startConnection() {
    //Step 6: Start a Connection
    try {
        tCon.start();
    } catch ( JMSException e) {
        System.err.println("Cannot start
            TopicConnection :" + e.toString());
        System.exit(1);
    }
}
```

메소드를 추가 시키기 위해서, connection 에 대한 사전 확인 작업이 필요하다.

```
public void startConnection() {
    try {
```

```
        //Step 6: Start a Connection
        if ( tCon != null ) {
            tCon.start();
        }
    } catch ( JMSException e) {
        System.err.println("Cannot start
            TopicConnection :" + e.toString());
        System.exit(1);
    }
}
```

위 어플리케이션 코드는 안정성을 유지하고 많은 시스템 자원을 소비하지 않으면서 connection 을 시작하는, 좋은 예를 보여주고 있다.

참고: 만일 메시지를 보내기만을 원한다면 connection 을 시작 할 필요는 없다. 다시 말해서, connection 을 시작하지 않고도 JEUS JMS Server 로 메시지를 보낼 수는 있지만 메시지를 받을 수는 없다. Connection 과 관련된 JMS 1.1 스펙을 참조하기 바란다.

7.3.5 Queue 에 메시지 보내기

다음 메소드는 queue 로 메시지를 보내는 것을 보여준다.

```
public void sendToQueue() {
    //Step 7: Do some application-specific operations
    TextMessage textMsg = null;
    try {
        textMsg = qSes.createTextMessage("Hello
            World!");
    } catch (JMSException ex) {
        System.err.println("Cannot create
            TextMessage :" + ex.toString());
        System.exit(1);
    }
    try {
        qsender.send(textMsg);
    } catch ( JMSException ex) {
        System.err.println("Cannot send
            TextMessage :" + ex.toString());
        System.exit(1);
    }
```

```
}
```

7.3.6 Topic 에 메시지 보내기

다음 메소드는 topic 에 메시지를 보내기 위한 것이다.

```
public void publishToTopic() {
    //Step 7: Do some application-specific operations
    TextMessage textMsg = null;
    try {
        textMsg = tSes.createTextMessage("Hello
            World!");
    } catch (JMSException ex) {
        System.err.println("Cannot create
            TextMessage :" + ex.toString());
        System.exit(1);
    }
    try {
        tpublisher.publish(textMsg);
    } catch ( JMSException ex) {
        System.err.println("Cannot publish
            TextMessage :" + ex.toString());
        System.exit(1);
    }
}
```

7.3.7 리소스 종료하기

클라이언트 어플리케이션은 connection 과 session 같은 리소스를 사용하고 나면 close 시켜서 리소스를 해제해 준다. connection 을 닫으면, 그 connection 에서 사용한 모든 session 도 닫히게 된다. 그러나 그 반대는 성립하지 않는다. session 을 닫으면 동기/비동기적으로 메시지 받는 작업만 종료된다. 그러므로 반드시 어플리케이션의 끝에는 connection 을 닫아준다.

Queue 의 경우에 다음 releaseReousrce() 메소드는 connection 을 끝내기 위한 것이다. Topic 의 경우에도 똑같이 적용이 된다.

```
public void releaseResource() {
    try {
        //Step 7: Close a Connection
        qCon.close();
    } catch (JMSException ex) {
```

```
        System.err.println("Cannot close
            QueueConnection :" + ex.toString());
    }
}
}
```

앞서 언급한 것과 같이, 실제 어플리케이션에서는 성능을 고려하지 말고 해당 connection 을 체크해야 한다.

```
public void releaseResource() {
    try {
        //Step 8 Close a Connection
        if ( qCon != null ) {
            qCon.close();
        }
    } catch (JMSException ex) {
        System.err.println("Cannot close
            QueueConnection :" + ex.toString());
    }
}
```

7.3.8 전체적인 어플리케이션 구조

Queue 에 메시지를 보내거나 Topic 에 메시지를 보내는 간단한 어플리케이션을 살펴보겠다. 전체적인 어플리케이션은 Queue 의 경우에 대해서 다음과 같이 설명한다. Topic 의 경우도 동일하기 때문에 생략한다.

<<qSender.java>>

```
public class qSender {
    public qSender() {}
    public void initResource() {
        . . .
    }
    public void startConnection() {
        . . .
    }
    public void sendToQueue() {
        . . .
    }
    public void releaseResource() {
```

```
        . . .
    }
    public static void main(String[] args) {
        qSender sender = new qSender();
        sender.initResource();
        sender.startConnection();
        sender.sendToQueue();
        sender.releaseResource();
    }
}
```

만일 JDK 가 Runtime.addShutdownHook() 메소드를 지원한다면, 다음과 같이 수정할 수 있다.

<<qSender.java>>

```
public class qSender {
    . . .
    public static void main(String[] args) {
        final qSender sender = new qSender();
        sender.initResource();
        sender.startConnection();
        sender.sendToQueue();
        Runtime.getRuntime().addShutdownHook(
          new Thread() {
                public void run() {
                    sender.releaseResource();
                }
          }
        );
    }
}
```

다음은 윈도우즈 플랫폼에서 어플리케이션을 실행하기 위한 스크립트이다. tSubscriber 경우에도 유사하다.

```
C:\jeus50\samples\ manual_examples\jmsguide\bin>type qSender.bat
java -
classpath %JEUS_HOME%\lib\system\jeus.jar;%JEUS_HOME%\lib\system\jmxr
i.jar;%JEUS_HOME%\lib\system\jmxremote.jar;%JEUS_HOME%\lib\system\jmx
tools.jar;%JEUS_HOME%\lib\system\jaxb-
impl.jar;%JEUS_HOME%\lib\system\jaxb-
```

```
libs.jar;%JEUS_HOME%\lib\system\jaxb-
api.jar;%JEUS_HOME%\lib\system\relaxngDatatype.jar;%JEUS_HOME%\lib\sy
stem\xsdlib.jar;%JEUS_HOME%\lib\system\xml_resource.jar;%JEUS_HOME%\l
ib\system\jxml-impl.jar;C:\jeus50\samples\jms\classes -
Djava.naming.factory.initial=jeus.jndi.JEUSContextFactory -
Djeus.baseport=%JEUS_BASEPORT% sample.manual.qSender
```

7.3.9 결론

이상으로 일반적인 JMS 어플리케이션 구조를 소개하고, JMS 어플리케이션이 어떻게 구성이 되는지를 살펴 보았다. 다음 절에서 다양한 어플리케이션에서 메시지를 어떻게 받는지에 대해서 살펴본다.

7.4 Message 받기

7.4.1 소개

이 절에서 동기/비동기적으로 메시지를 받는 두 가지 예제를 살펴본다. Queue destination 에 대해서 동기적으로 메시지를 받고 Topic destination 에 대해서 비동기적으로 메시지를 받는 것에 대해서 살펴본다. JMS 스펙에는 동기적으로 메시지를 받는것을 queue 그리고 비동기적으로 메시지를 받는 것을 topic 으로 엄격히 제한을 두지않고 있다. 예제는 일반적인 JMS 어플리케이션 구조를 따르고 있다. 이전 7.3 절과 매우 유사하다.

7.4.2 Queue 으로부터 메시지 받기를 위한 리소스 설정

우선, Queue 로부터 동기적으로 message 를 받는 클래스의 이름을 qReceiver 라고 부르고, 이 qReceiver 클래스는 다음과 같이 필요한 패키지를 import 한다.

Queue 로부터 메시지를 받는 qReceiver 클래스를 실행한다. 다음 코드는 필요한 패키지와 클래스 변수를 import 하는것을 보여준다.

<<*qReceiver.java*>>

```
package sample.mannual;
import javax.naming.*;
import javax.jms.*;
  public class qReceiver {
      Context ctx = null;
      QueueConnectionFactory qConFactory = null;
      QueueConnection qCon = null;
```

```
      QueueSession qSes = null;
      Queue queue = null;
      QueueReceiver qreceiver = null;
       ...
```

그러면, initResource() 메소드에서 JMS 리소스로 생각되는 클래스 변수들을 초기화 한다.

```
public void initResource() {
      try {
          ctx = new InitialContext();
      } catch (NamingException ex) {
          System.err.print("Cannot create Context :"
              + ex.toString() );
          System.exit(1);
      }
      //Step 1: Lookup a ConnectionFactory
      try {
          qConFactory = (QueueConnectionFactory)
              ctx.lookup("QueueConnectionFactory");
      } catch (NamingException ex) {
          System.err.print("Cannot find
              QueueConnectionFactory in JNDI :" +
              ex.toString());
          System.exit(1);
      }
      //Step 2: Create a Connection
      try {
          qCon = qConFactory.createQueueConnection();
      } catch ( JMSException ex) {
          System.err.println("Cannot create
              QueueConnection :" + ex.toString());
          System.exit(1);
      }
      //Step 3: Create a Session
      try {
          qSes = qCon.createQueueSession(
              false,Session.AUTO_ACKNOWLEDGE );
      } catch (JMSException ex) {
          System.err.println("Cannot create
```

```
                QueueSession :" + ex.toString());
        }
        //Step 4: Lookup a Destination
        try {
              queue = (Queue)ctx.lookup("ExamplesQueue");
        } catch (NamingException ex) {
              System.err.println("Cannot find ExamplesQueue
              in JNDI :" + ex.toString());
        }
        //Step 5: Create a  MessageProducer or a
              MessageConsumer, or register a
              MessageListener
        try {
              qreceiver = qSes.createReceiver(queue);
        } catch( JMSException ex) {
              System.err.println("Cannot create
              QueueReceiver :" + ex.toString());
              System.exit(1);
        }
  }
```

7.4.3 Topic 으로부터 메시지 받기를 위한 리소스 설정

Topic 의 경우에 비동기적으로 메시지를 받는 클래스 이름을 tSubscriber 라고 한다.

아래 Topic 예제에서는 모니터 변수가 있는 것이 이전의 Queue 예제와 차이가 있다. 이 예제와 같이 async 하게 메세지를 받는 MessageListener 를 등록하는 경우에는, 특정 이벤트를 기다리는 모니터 변수가 main program 에서 사용된다.

예를 들어 non-text message 가 도착하면 그것을 통지 받고서 main thread 에게 상태를 알려준다.

tSubscriber 클래스는 비동기적으로 MessageListener 를 등록하기 위해서 MessageListener interface 를 구현한다.

<<tSubscriber.java>>

```
package sample.mannual;
import javax.jms.*;
import javax.naming.*;
```

```
public class tSubscriber implements MessageListener {
    Context ctx = null;
    TopicConnectionFactory tConFactory = null;
    TopicConnection tCon = null;
    TopicSession tSes = null;
    Topic topic = null;
    TopicSubscriber subscriber = null;
    private boolean[] monitor = new boolean[1];
    public tSubscriber() {
        monitor[0] = false;
    }
    public void onMessage(javax.jms.Message msg) {
       . . .
    }
```

그러면, 다음과 같이 initResources() 메소드에서 JMS 리소스로 사용될 클래스 변수를 초기화 한다. 비동기적으로 메시지를 받도록 하기 위해서, topic subscriber 에 MessageListener 를 등록한다.

```
public void initResource() {
    try {
        ctx = new InitialContext();
    } catch (NamingException ex) {
        System.err.print("Cannot create Context :" +
            ex.toString() );
        System.exit(1);
    }
    //Step 1: Lookup a ConnectionFactory
    try {
        tConFactory = (TopicConnectionFactory)
            ctx.lookup("TopicConnectionFactory");
    } catch (NamingException ex) {
        System.err.print("Cannot find
            TopicConnectionFactory in JNDI :" +
            ex.toString());
        System.exit(1);
    }
    //Step 2: Create a Connection
    try {
        tCon = tConFactory.createTopicConnection();
```

```
    } catch ( JMSException ex) {
        System.err.println("Cannot create
            TopicConnection :" + ex.toString());
        System.exit(1);
    }
    //Step 3: Create a Session
    try {
        tSes = tCon.createTopicSession(
            false,Session.AUTO_ACKNOWLEDGE );
    } catch (JMSException ex) {
        System.err.println("Cannot create
            TopicSession :" + ex.toString());
        System.exit(1);
    }
    //Step 4: Lookup a Destination
    try {
        topic = (Topic)ctx.lookup("ExamplesTopic");
    } catch (NamingException ex) {
        System.err.println("Cannot find ExamplesTopic
            in JNDI :" + ex.toString());
        System.exit(1);
    }
    //Step 5: Create a  MessageProducer or a
        MessageConsumer, or register a
        MessageListener
    try {
        subscriber = tSes.createSubscriber(topic);
    } catch (JMSException ex) {
        System.err.println("Cannot create
            Subscriber :" + ex.toString());
        System.exit(1);
    }
    try {
        subscriber.setMessageListener(this);
    } catch (JMSException ex) {
        System.err.println("Cannot register
            MessageListener :" + ex.toString());
        System.exit(1);
    }
```

```
}
```

7.4.4 Connection 시작

Connection 은 initResource() 메소드에서 초기화가 된다. 그런 후 startConnection() 메소드에서 실행된다. 다음은 Topic 경우의 예제이지만 Queue 도 똑같이 동작한다.

```
public void startConnection() {
    try {
        //Step 6: Start a Connection
        tCon.start();
    } catch ( JMSException e) {
        System.err.println("Cannot start
            TopicConnection :" + e.toString());
        System.exit(1);
    }
}
```

7.4.5 동기적으로 메시지 받기

다음 메소드는 queue 로부터 message 를 받는 것을 보여준다.

```
public void receiveFromQueue() {
    Message msg = null;
    //Step 7: Do some application-specific operations
    try {
        msg = qreceiver.receive();
    } catch ( JMSException ex) {
        System.err.println("Cannot send TextMessage
            :" + ex.toString());
        System.exit(1);
    }
    System.out.println("Received message : " + msg);
}
```

7.4.6 비동기적으로 메시지 받기

다음 코드 일부분은 onMessage 메소드의 구현을 보여준다. 메시지가 도착을 하면 그 메소드는 자동적으로 실행한다.

```
public void onMessage(javax.jms.Message msg){
```

```
    //Step 7: Do some application-specific operations
    if ( msg instanceof TextMessage ) {
        System.out.println("TextMessage arrived :"
            + msg);
    } else {
        System.out.println("Non-TextMessage arrived :
            " + msg + ", so stop listening");
        allDone();
    }
}
```

특히 onMessage 가 실행될 상황이 되면, 메인 프로그램 특히 메인 쓰레드에 이를 통보하도록 디자인 되어있다. onMessage 메소드를 실행시키는 thread 는 connection factory 의 client 측의 thread pool 중의 하나이다. Client 측에서 thread pool 은 JEUS JMS Server 측의 connection factory descriptor 나 JVM 파라미터에서 설정된다. 이에 관해서는 4.5 절을 참조한다.

MessageListener 를 등록해 놓은 경우, 이것을 등록한 쓰래드에서 connection 을 close 할 수 있다. 그러나 onMessage 메소드에서 connection 을 close 하는 경우가 일반적이다.

예를 들어, 마지막 메세지까지 받아야 하는 어플리케이션이라면 마지막 메세지 받기 전까지 클라이언트 어플리케이션에서 connection 을 끊지 않고 유지해야 한다. 클라이언트 어플리케이션에서는 끝나야 할 조건이 되었다면 (onMessage 수행 중 마지막 메시지라고 판단될 때) connection 을 close 한다.

AllDone 메소드는 다음 같다.

```
private void allDone() {
    synchronized (monitor) {
        monitor[0] = true;
        monitor.notify();
    }
}
```

메인 쓰레드(이 경우에는 메인 메소드가 실행 중인 쓰레드)는 onMessage() 메소드 쓰레드가 모니터 변수를 notify 하는 것을 기다린다. 이런 내용은 다음 waitTillDone() 메소드에서 처리된다.

```
private void waitTillDone() {
    synchronized (monitor) {
```

```
        while (! monitor[0]) {
            try {
                monitor.wait();
            } catch (InterruptedException ie)
            {}
        }
    }
}
```

7.4.7 리소스 해제

리소스를 해제하는 것 7.3.7 절에서와 같은 방법으로 실행한다. releasResource() 메소드는 Topic 의 경우에 대해서 다음과 같다.

```
public void releaseResource() {
    try {
        //Step 8 Close a Connection
        tCon.close();
    } catch (JMSException ex) {
        System.err.println("Cannot close
            TopicConnection :" + ex.toString());
    }
}
```

7.4.8 전체적인 어플리케이션 구조

Queue 로부터 동기적으로 메시지를 받는것과 Topic 으로부터 비동기적으로 메시지를 받는 두가지 경우에 대해 살펴 보았다. 전체적인 어플리케이션은 다음 코드처럼 실행이 된다.

Queue 의 경우

<<*qReceiver.java*>>

```
public class qReceiver {
    . . .
    public void initResource() {
        . . .
    }
    public void startConnection() {
        . . .
    }
    public void receiveFromQueue() {
```

```
        . . .
    }
    public void releaseResource() {
        . . .
    }
    public static void main(String[] args) {
        qReceiver receiver = new qReceiver();
        receiver.initResource();
        receiver.startConnection();
        receiver.receiveFromQueue();
        receiver.releaseResource();
    }
}
```

Topic 의 경우

<<tSubscriber.java>>

```
public class tSubscriber {
    . . .
    public void initResource() {
        . . .
    }
    public void startConnection() {
        . . .
    }
    public void receiveFromQueue() {
        . . .
    }
    public void releaseResource() {
        . . .
    }
    public void onMessage(javax.jms.Message msg) {
        . . .
        allDone();
    }
    private void waitTillDone() {
        . . .
        monitor.wait();
    }
```

```
    private void allDone() {
        . . .
         monitor.notify();
    }
    public static void main(String[] args) {
        tSubscriber subscriber = new tSubscriber();
        subscriber.initResource();
        subscriber.startConnection();
        subscriber.waitTillDone();
        subscriber.releaseResource();
    }
}
```

tSubscriber 클래스를 위한 Windows 용 script 는 다음과 같다. 이것은 qReceiver 경우와도 같다.

```
C:\jeus50\samples\ manual_examples\jmsguide\bin>type tSubscriber.bat
java -
classpath %JEUS_HOME%\lib\system\jeus.jar;%JEUS_HOME%\lib\system\jmxr
i.jar;%JEUS_HOME%\lib\system\jmxremote.jar;%JEUS_HOME%\lib\system\jmx
tools.jar;%JEUS_HOME%\lib\system\jaxb-
impl.jar;%JEUS_HOME%\lib\system\jaxb-
libs.jar;%JEUS_HOME%\lib\system\jaxb-
api.jar;%JEUS_HOME%\lib\system\relaxngDatatype.jar;%JEUS_HOME%\lib\sy
stem\xsdlib.jar;%JEUS_HOME%\lib\system\xml_resource.jar;%JEUS_HOME%\l
ib\system\jxml-impl.jar;C:\jeus50\samples\jms\classes -
Djava.naming.factory.initial=jeus.jndi.JEUSContextFactory -
Djeus.baseport=%JEUS_BASEPORT% sample.manual.tSubscriber
```

7.4.9 결론

이번 절에서 JEUS JMS 에서는 메시지 받는 프로그램을 어떻게 개발해야 하는지를 설명했다. 동기적으로 메시지를 받는 것은 JEUS JMS 에서 간단하게 실행될 수 있다. 반면 비동기적으로 메시지 받는 것은 쓰레드 프로그래밍이 필요하다.. 다음 절에서 session acknowlede 가 어떻게 사용 되는지를 설명한다.

7.5 Message 의 Acknowledge 처리

session 의 acknowledge 모드가 Session.CLIENT_ACKNOWLEDGE 이면, 클라이언트 어플리케이션에서 아래와 같은 acknowledge() 메소드를 호출해서, 받은 messages 에 대해서 명시적으로 acknowledge 처리를 해야 한다.

<<*Message.java*>>

```
public interface Message {
    . . .
    public void acknowledge() {} throws JMSException
    . . .
}
```

.

만일 메시지를 받는 session 이 `Session.CLIENT_ACKNOWLEDGE` 모드가 아니라면, `Session.AUTO_ACKNOWLEDGE` 나 `Session.DUPS_OK_ACKNOWLEDGE`, 아니면 트랜잭션을 사용하는 경우이다(트랜잭션의 경우 클라이언트 acknowledge 는 무시된다).

만일 모드가 `Session.AUTO_ACKNOWLEDGE` 거나 `Session.DUPS_OK_ACKNOWLDEGE` 이면, `javax.jms.Message.Consumer.receive()` 메소드가 실행되거나 `javax.jms.MessageListener.onMessage()` 메소드가 리턴이 될 때마다 자동적으로 acknowledge 가 실행된다.

트랜잭션을 사용하는 session 의 경우에 `session.commit()`은 일반적인 룰을 따라서 실행된다.

참고: 트랜잭션 session 에 관련해서는 7.11 절을 참조한다.

참고: `MessageConsumer.receive()` 메소드는 `MessageConsumer.receive()`, `MessageConsumer.receive(long timeout)`, `MessageConsumer.receiveNoWait()`들이 있다. 자세한 것은 JMS 1.1 스펙이나 API 문서를 참조한다.

`Message.acknowledge()` 메소드는 메시지를 acknowledge 하기 전이든 후이든 상관하지 않고 session 이 받은 모든 메시지에 대해 영향을 미친다. 간략하게 말하자면, acknowledge 은 session 레벨의 활동이다. 이런 topic 과 관련된 내용은 JMS 1.1 스펙을 참조한다.

Not-acknowledge 된 메시지는 `javax.jms.Session.recover()` 메소드 또는 `javax.jms.MessageConsumer.receiver()` 메소드가 호출이 될 때 재전송이

되거나, queue receiver 또는 durable subscriber 가 만들어질 때 메시지가 재전송이 된다.

다음 예제 코드는 어플리케이션에서 acknowledge 가 어떻게 실행이 되는지를 보여준다.

<<qAsyncReceiverClientAck.java>>

```
public class qAsyncReceiverClientAck
    implements MessageListener {
    ...
    public void initResource() {
        ...
        qSes = qCon.createQueueSession(
            false,Session.CLIENT_ACKNOWLEDGE);
        ...
        qreceiver.setMessageListener(this);
        ...
    }
    public void onMessage(javax.jms.Message msg){
        try {
            if ( msg instanceof TextMessage ) {
                if ( !msg.getJMSRedelivered() ) {
                    System.out.println("[1st] TextMessage
                        arrived :" + msg);
                } else {
                    System.out.println("[2nd] TextMessage
                        arrived :" + msg);
                }
            } else {
                if ( !msg.getJMSRedelivered() ) {
                    System.out.println("[1st] Non-
                        TextMessage arrived :" + msg + ",
                        so recover unacknowledged
                        messages");
                    qSes.recover();
                } else {
                    System.out.println("[2nd] Non-
                        TextMessage arrived :" + msg + ",
                        so stop listening");
                    msg.acknowledge();
```

```
                allDone();
            }
        }
    } catch( JMSException e ) {
        System.err.println("Error in handling onMessage
            with " + msg.toString() + ": " +
            e.toString());
        allDone();
    }
}
...
```

7.6 Message Recovery

7.6.1 소개

Message Recovery 는 unacknowledge 된 메시지를 재전송 하도록 한다. 이 절에서는 다음 항목에 따라서 메시지를 복구하는 것에 대해서 다룬다.

- onMessage 에서 auto-recovery 처리하기
- Recovery 지연 시간 처리하기
- Recovery 제한 시간 처리하기

7.6.2 onMessage 에서 auto-recovery 처리하기

비동기적으로 메시지를 받도록 등록이 되어있는 경우, onMessage 메소드는 클라이언트 측 connectionfactory thread pool 의 thread 에서 실행이 된다.

만일 session ack-mode 가 Session.AUTO_ACKNOWLEDGE 또는 Session.DUPS_OK_ACKNOWLEDGE 이거나, onMessage 를 실행할 동안 RuntimeException 이 발생했다면, 메시지가 기본적으로 재전송된다. JMS 어플리케이션은 다음 설정에 의해서 이 auto-recovery 속성을 사용할 수 있다.

- `-Djeus.jms.client.session.autoRecoverOnFail=<true/false>`, 디폴트 값은 true.

7.6.3 Recovery 지연 시간 처리하기

jeus.jms.common.jms.session.SessionImpl 는 사용자 어플리케이션에게 다음의 메소드를 통해서 복구 지연 시간(Recovery Delay Time)을 설정할 수 있다. 지정한 시간 후에 Recovery 된 메시지가 재전송된다.

<<*SessionImpl.java*>>

```
public class SessionImpl {
    ...
    public void recover(long _delaytime)
        throws JMSException {
    }
    ...
}
```

'_delaytime' 파라미터는 0 보다 더 커야 하고 millisecond 단위로 표현이 되어야 한다. 만일 0 보다 크지 않다면, JMSException 을 던진다. 메소드는 지정한 delaytime 시간 동안 블록되지는 않지만, 성공하면 리턴한다.

JMS 클라이언트 어플리케이션에 대한 기본적인 Session Recovery 대기 시간은 다음 JVM 파라미터를 사용해서 설정할 수 있다.

- -Djeus.jms.client.session.DefaultDelayTimeOnRecover=<delay time in millisecond>, 디폴트 값은 –1 이다. 이것은 delay 되지 않는 것을 의미한다.

7.6.4 Recovery 제한 시간 처리하기

사용자 어플리케이션에서 무한정으로 Recovery 하는 것을 막기위해 Recovery 값을 세팅할 수 있다. 이것은 사용자 어플리케이션이 연속해서 `Session.recover()` 메소드를 호출할 때 자주 발생한다.

- -Djeus.jms.client.session.LimitForRecover=<Limit count for recover>, Default 값은 –1 이다. 만일 –1 로 세팅을 한다면 Recovery 개수의 제한이 없다는 것을 의미한다.

만일 `Sessin.recover()` 또는 `SessionImpl.recover(long delaytime)`를 제한 회수 이상으로 호출하면, JMSError 가 발생한다.

7.6.5 결론

클라이언트 어플리케이션은 unacknowledge 메시지를 session 에 재전송 하도록 하기 위해서 `Sessin.recover()`를 사용할 수 있다. JEUS JMS API 는 Session Recovery 에 관련된 환경을 제공한다.

7.7 Message 정보 세팅

7.7.1 소개

JEUS JMS Message 는 메시지 바디, 속성, 표준 헤더 필드를 지원한다. 메시지가 전송될 때, 메시지 바디, 속성, 헤더 정보 등 전체 내용이 보내진다. 이번 절에서는 이 내용에 대해서 설명한다.

7.7.2 Message 헤더의 세팅

다음 리스트들은 메시지 헤더에 대한 set/get 메소드를 포함하는 메시지 헤더들이다.

다음은 클라이언트 어플리케이션에 set 되는 헤더들 이다.

- **JMSCorrelationID**
- **JMSReplyTo**
- **JMSType**

다음은 `MessageProducer.sendMessage()` 메소드가 리턴된 후에 자동으로 세팅이 되는 헤더들이다.

- **JMSDestination**
- **JMSDeliveryMode**
- **JMSExpiration**
- **JMSPriority**
- **JMSMessageID**
- **JMSTimestamp**

참고: 클라이언트 어플리케이션은 `MessageProducer.sendMessage()` 메소드 이전이든 이후이든 명시적으로 JMSMessageID 를 설정 할 수는 있지만, 의미가 없다.

다음은 `MessageConsumer.receive()` 메소드가 리턴한 후에 또는 `MessageListener.onMessage()`실행 후에 자동으로 설정이 되는 헤더들이다.

- **JMSRedelivered**

다음 예제를 살펴보기 바란다.

```
public void sendToQueue() {
    TextMessage textMsg = null;
    try {
        textMsg = qSes.createTextMessage(
            "Hello World!");
        System.out.println(“before sending,
            JMSMessageID:” +
            textMsg.getJMSMessageID());
        //before sending, JMSMessageID: null
    } catch (JMSException ex) {
        System.err.println("Cannot create
            TextMessage :" + ex.toString());
        System.exit(1);
    }
    try {
        qsender.send(textMsg);
        System.out.println("after sending,
            JMSMessageID:" +
            textMsg.getJMSMessageID());
        //after sending, JMSMessageID: ID:21
    } catch ( JMSException ex) {
        System.err.println("Cannot send
            TextMessage :" + ex.toString());
        System.exit(1);
    }
}
```

7.7.3 Message 속성 세팅

메시지 속성을 write 하기 위해서는 속성에 대한 set 메소드가, 그리고 read 하기 위해서는 get 메소드가 사용이 되어야 한다. 일반적으로, 메시지 property 은 MessageProducer 에서 설정되고, MessageConsumer 나 MessageListener 에서 보여진다.

다음 테이블은 write 와 read 사이에서 속성값의 변화를 보여준다.[표 1] .

표 1.write 또는 read 할 때의 속성 변환

Property to write as	Property readable as								
	boolean	byte	short	int	long	float	double	String	Object
boolean	x							x	x
byte		x	x	x	x			x	x
short			x	x	x			x	x
int				x	x			x	x
long					x			x	x
float						x	x	x	x
double							x	x	x
String	x	x	x	x	x	x	x	x	x
Object	x	x	x	x	x	x	x	x	x

`SetObjectProperty()` 메소드는 단지 NULL 값 또는 Boolean, Byte, Short, Integer, Long, Float, Double, String 값들만 받는다는 것을 유념하기 바란다. 다음 코드은 어떻게 메시지 속성 변환이 일어나는지를 보여준다.

```
TextMessage message = session.createTextMessage(
```

```
    "test");
message.setBooleanProperty("name1", false);
message.setDoubleProperty("name2", 123.456789e222);
message.setIntProperty("name3", 223344);
try {
    System.out.println("try to write invalid object
        type by writeObject");
    message.setObjectProperty("name4", new Vector());
    System.out.println("No exception occurred,
        error");
} catch (JMSException ex)  {
    System.out.println("Expected exception occurred "
        + ex);
}
message.setObjectProperty("name4", new
    Integer(223344));
System.out.println("Reading property items of various
    data types as String:");
System.out.println(" Boolean: " +
    message.getStringProperty("name1"));
System.out.println(" Double: " +
    message.getObjectProperty("name2"));
System.out.println(" Int: " +
    message.getStringProperty("name3"));
System.out.println(" Int by writeObject: " +
    message.getStringProperty("name4"));
System.out.println(" Int by writeObject
    with getIntProperty : " +
    message.getIntProperty("4"));
```

Boolean, Byte, Short, Integer, Long, Float, Double, String 값들이 쓰일 때, 그값들은 타깃 메시지 인스턴스로 복사된다. 그래서 참조변수를 재사용할 수 있다.

다음 코드를 살펴보자.

```
String value = “This is from the earth”;
message.setStringProperty("name5", value);
value = “This is from the world”;
message.setStringProperty("name6", value);
```

```
System.out.println(" String: " +
    message.getStringProperty("name5"));
// String: This is from the earth
System.out.println(" String: " +
    message.getStringProperty("name6"));
// String: This is from the world
```

JMS 클라이언트 어플리케이션이 메시지를 받았을 때, 메시지의 속성값들은 단지 읽을 수만 있다. 만일 어플리케이션이 그 속성값을 수정하려고 한다면, 다음 `Message.clearProperties()` 메소드를 사용 해야 한다.

<<*Message.java*>>

```
public interface Message {
    ...
    void clearProperties() throws JMSException;
    ...
}
```

7.7.4 Message Body 설정

JEUS JMS 클라이언트 어플리케이션이 이용할 수 있는 메시지 타입으로는 6 종류가 있다. BytesMessage, MapMessage, ObjectMessage, StreamMessage, TextMessage, Message 가 그것이다. 속성이 타깃 메시지에 복사되는 것처럼, 메시지 바디 내용도 메시지에 복사되어, 재사용될 수 있다.

```
String content = “1st message”;
TextMessage message1 = session.createTextMessage(
    content);
content = “2nd message”;
TextMessage message2 = session.createTextMessage(
    content);
System.out.println(“message1:” + message1.getText());
//message1:1st message
System.out.println(“message1:” + message2.getText());
//message1:2nd message
```

7.7.5 결론

메시지는 헤더, 속성 그리고 바디로 구성이 된다. 어떤 헤더 값들은 JMS 클라이언트 어플리케이션에 의해서 설정이 될 수 있고 다른 값은 JEUS JMS 에 의해 설정이 된다. 속성과 바디는 또한 value-copy 방법으로 JMS 클라이언

트 어플리케이션에 의해서 설정이 될 수 있다. 다음 절에서는 선택한 메시지만을 어떻게 골라서 받아들이는지에 대해 설명한다.

7.8 Message 필터링

Message selector 은 메시지를 필터링하는데 사용할 수 있고 필터링 기준을 충족 시키는 메시지만을 선택 하는데 사용할 수 있다.

Message selector 는 `QueueSession.createReceiver(Queue queue, String selector)` 또는 `TopicSession.createSubscriber(Topic topic, String selector, boolean noLocal)`를 사용해서 queue receiver 나 topic subscriber 을 만드는데 설정할 수 있다.

해당 Queue 또는 Topic 에 대해서 Message Selector 는 JEUS JMS Server 측에서 처리한다. 그래서, 선택되지 않는 message 는 네트워크의 혼잡을 줄이기 위해서 QueueReceiver 또는 TopicSubscriber 에 전송이 되지 않는다.

만일 Selector 값이 NULL 또는 TRUE 이면, 내부적으로 무시된다. 빈 괄호는 조건의 범위를 명확하게 하는데는 도움이 되지만, 서버 쪽의 오버헤드를 줄이기 위해서 사용하지 않기를 권한다. 예를 들어, (((name = 'James') or (age > 30)) and ((height < 170))) 보다는 (name = 'James' or age > 30) and height < 170 을 권한다.

다음은 selector 예제이다.

- **manager = 'Vialli'**

```
message.setStringProperty("manager", "Vialli");
```

- **gender = 'M' AND salary > 100**

```
message.setStringProperty("gender", "M");
message.setIntProperty("salary", 120 );
```

- **gender = 'M' OR salary > 100**

```
message.setStringProperty("gender", "M");
message.setIntProperty("salary", 60 );
```

- **name in ('all', 'Bill', 'John')**

```
message.setStringProperty("name", "Bill");
```

- **JMSType = ‘food’**

```
message.setJMSType(“food”);
```

- **weddingyear between 1992 and 2002 or name is not null**

```
message.setIntProperty(“weddingyear”,2001);
message.setStringProperty(“name”,”Richard”);
```

7.9 Destination 만들기

7.9.1 소개

이 절에서 우리는 JMS 클라이언트 어플리케이션에서 참조하는 destination 을 얻은 방법에 대해서 설명한다. 일반적으로, 타깃 destination 은 JNDI lookup 을 통해서 얻을 수 있다. destination 을 얻거나 만드는데는 여러 방법들이 있다.

7.9.2 Destination 얻기

메시지를 보내거나 또는 메시지를 받는 타깃 destination 은 JNDI lookup 을 통해서 혹은 Session 레벨의 메소드에 의해서 얻을 수 있다.

JNDI lookup 을 사용하는 방법은 7.3.2 절에서 언급된 대로 실행할 수 있다.

`javax.jms.QueueSession` 은 `createQueue()` 메소드를 가지고 있고 `javax.jms.TopicSession` 은 `createTopic()` 메소드를 가지고 있다. 각 메소드는 JMS Server 측에서 지정된 이름의 destination 을 찾고, destination 의 레퍼런스를 리턴한다. 만일 server 측에서 존재하지 않는 다면 JMSException 을 발생시킨다.

<<*QueueSession.java*>>

```
public interface QueueSession {
    ...
    public Queue createQueue(String queueName)
        throws JMSException;
    ...
}
public interface TopicSession {
    ...
    public Topic createTopic(String topicName)
        throws JMSException;
```

```
    ...
}
```

참고: 위의 `createQueue()`와 `createTopic()` 메소드는 동적으로 destination 을 만들지 못한다. 단지 그것의 참조를 얻을 뿐이다. 동적으로 destination 을 만드는 것과 관련된 내용은 7.9.3 절을 본다.

7.9.3 Temporary Destination 의 사용

Temporary destination 은 JMS 클라이언트 어플리케이션 내에서 만들어 질 수 있다. 그러나 temporary destination 을 만든 connection 의 사용 기간 동안만 사용이 가능하다. 일반적으로 JMSReplyTo 헤드 필드 등에서 Temporary destination 이 사용된다. 만들어진 Temporary Destination 을 더 이상 사용하지 않을 때에는 JEUS JMS Server 로부터 delete 메소드를 사용해 지울 수 있다.

Temporary Queue 을 만들기 위해서 다음의 메소드를 사용한다.

<<*QueueSession.java*>>

```
public interface QueueSession {
    ...
    TemporaryQueue createTemporaryQueue()
        throws JMSException;
    ...
}
```

Temporary Topic 을 만들기 위해서 다음 메소드를 사용한다.

<<*TopicSession.java*>>

```
public interface TopicSession {
    . . .
    TemporaryTopic createTemporaryTopic()
        throws JMSException;
    . . .
}
```

temporary destination 으로부터 만들어진 MessageConsumer 또는 MessageListener 는 메시지를 받을 수는 있지만 보낼 수는 없다.

그러나, 다른 connection 으로부터 생성된 MessageProducer 은 그곳에 메시지를 보낼 수 있다. 이것은 일반적으로 JMSReplyTo 헤더에 temporary destination 을 설정해서 가능하다.

JMSReplyTo 의 경우와 다른 방법으로 그곳으로 메시지를 보내기 위해서는, JMS 어플리케이션이 7.9.2 절에서 언급된 것과 같이 temporary destination 의 레퍼런스를 얻어야 한다. TemporaryQueue 또는 TemporaryTopic 의 toStirng 메소드는 export-name 을 리턴한다.

`getQueueName()` 또는 `getTopicName()` 메소드는 각각 `QueueSession.createQueue()` 메소드 또는 `TopicSession.createTopic()` 메소드가 생성한 destination 이름을 보여준다.

다음 코드는 queue 의 경우 temporary destination 을 어떻게 사용한는지 보여준다.

<<TempQueue.java>>

```
public class TempQueue {
    ...
    QueueSession session = qcon.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE);
    Queue tempQueue = session.createTemporaryQueue();
    System.out.println("check temporary queue by
        jms admin");
    System.out.println(" jndi:" +
        tempQueue.toString() + ", getQueueName: " +
        tempQueue.getQueueName() );
    QueueSender sender = session.createSender(
        tempQueue);
    QueueReceiver receiver = session.createReceiver(
        tempQueue);
    TextMessage text = session.createTextMessage(
        "HELLO CONTINENT... ");
    connection.start();
    sender.send(text);
    System.out.println("[Sender] sent JMSMessageID :
        " + text.getJMSMessageID() );
    Message msg = receiver.receive();
    System.out.println("[Receiver] received
        JMSMessageID : " + msg.getJMSMessageID () + ",
        " + ((TextMessage)msg).getText());
    ...
}
```

위의 코드는 실행하면 다음과 같이 보인다.

```
C:\jeus50\samples\manual_examples\jmsguide\bin >TempQueue.bat
C:\jeus50\samples\jms\bin> java -
classpath %JEUS_HOME%\lib\system\jeus.jar;%JEUS_HOME%\lib\syst
em\jmxri.jar;%JEUS_HOME%\lib\system\jmxremote.jar;%JEUS_HOME%\
lib\system\jmxtools.jar;%JEUS_HOME%\lib\system\jaxb-
impl.jar;%JEUS_HOME%\lib\system\jaxb-
libs.jar;%JEUS_HOME%\lib\system\jaxb-
api.jar;%JEUS_HOME%\lib\system\relaxngDatatype.jar;%JEUS_HOME%
\lib\system\xsdlib.jar;%JEUS_HOME%\lib\system\xml_resource.jar
;%JEUS_HOME%\lib\system\jxml-impl.jar; -
Djava.naming.factory.initial=jeus.jndi.JEUSContextFactory -
Djeus.baseport=%JEUS_BASEPORT% sample.manual.TempQueue
[ErrorMsgManager] Message Manager is initialized
check temporary queue by jms admin
 jndi:193.148.2.62:9741-Cl0Con0TQueue0, getQueueName:
Cl0Con0TQueue0
[Sender] sent JMSMessageID : ID:12
[Receiver] received JMSMessageID : ID:12, HELLO CONTINENT!
```

참고: 비록 message delivery mode 가 DeliveryMode.PERSISTENT 일지라도, temporary destination 으로 보내는 것은 DeliveryMode.NON_PERSISTENT 로 간주된다. 서버의 장애가 발생할 경우 temporary destination 의 그 자체의 특성에 의해서 temporary destination 에 메시지를 보내는 것을 보장하지 않는다.

7.9.4 결론

destination 레퍼런스를 얻기위한 여러 가지 방법을 살펴 보았다. 이런 방법으로는 동적으로 destination 을 만드는 것과 `createQueue()`, `createTopic()` 또는 temporary destination 을 만드는 것이 있다.

7.10 Durable Subscription 사용

7.10.1 소개

Durable subscription 은 unsubscription 이 될 때까지 보내진 메시지를 받는 작업이 보장되는 topic destination 에서 사용된다. Durable subscription 은 subscriber 가 close 될 때, 비활성화된 것으로 간주된다. Durable subscription 의 비활성화 동안 메시지를 저장해 두었다가, 활성화 상태일 때 durable subscription 에 메시지를 배달 한다.

7.10.2 Durable Subscription 의 설정

새로운 Durable subscription 을 시작하기 위해서는 ClientID 를 가지고 있는 topic connection 이 필요하다. topic connection 을 위한 Client ID 는 다음의 connection factory descriptor 나 `connection.setClientID()` 메소드를 통해서 설정된다.

<<*Connection.java*>>

```
public interface Connection {
    ...
    public Topic setClientID(String clientID)
        throws JMSException;
    ...
}
```

클라이언트 어플리케이션이 위의 메소드를 사용해서 clientID 을 설정하는 것과 connection factory descriptor 에서 clientID 를 설정하는 것은 동일하다.

만일 clientID 가 이미 connection factory descriptor 에 의해서 설정되었고, `setClientID()` 메소드의 파라미터 clientID 가 이미 설정된 clientID 와 동일하다면 `IllegalStateException` 을 발생시킨다. 또한 Connection 이 끊어져도 `IllegalStateException` 을 발생시킨다.

만일 JMS Server 가 중복된 clientID 을 발견한다면, setClientID() 메소드는 InvalidClientException 을 발생시킨다. 클라이언트 어플리케이션은 다음 메소드를 사용해서 connection 에 설정된 client ID 를 확인할 수 있다.

<<*Connection.java*>>

```
public interface Connection {
    ...
    public String getClientID() throws JMSException;
    ...
}
```

ClientID 가 설정된 connection 의 TopicSession 을 사용하면, 다음 두 개의 `createDurableSubscriber()` 메소드로 durable subscription 을 만든다.

<<*TopicSession.java*>>

```
public interface TopicSession {
    ...
    public TopicSubscriber createDurableSubscriber(
        Topic topic,
```

```
        String name
    ) throws JMSException;

    public TopicSubscriber createDurableSubscriber(
        Topic topic,
        String name,
        String messageSelector,
        boolean noLocal
    ) throws JMSException;
    ...
}
```

만일 클라이언트 어플리케이션이 durable subscription 을 재실행 한다면, 다시 말해서 Close 되어 있는 동안에 durable subscription 에 쌓인 메시지들을 받으려면, ClientID 와 이전에 세팅된 Topic 과 Message Selector 를 가진 Durable subscription 의 이름이 동일해야 하다.

만일 Topic 또는 message Selector 가 이전의 durable subscription 중의 하나와 같지 않다면, durable subscription 은 바뀌게 된다. 바뀐 durable subscription 은 이전에 존재하는 durable subscription 을 unsubscribing 시키고 새로운 durable subscription 을 등록하게 된다.

7.10.3 Durable Subscription 의 제거

durable subscription 이 실행되지 않을 때, 다시 말해서 존재하고 있는 durable subscriber 가 Close 되면, 다음 메소드를 사용해서 삭제할 수 있다. 만일 durable subscriber 가 실행 중이면 JMSException 을 발생시킨다.

<<*TopicSession.java*>>

```
public interface TopicSession {
    ...
    public void unsubscribe(String name)
        throws JMSException;
    ...
}
```

파라미터 `name` 은 `createDurableSubscriber()` 메소드에서 생성한 durable subscriber 이름이다.

7.10.4 결론

이 절에서는 durable subscription 을 어떻게 시작을 하고 끝내는지를 살펴 보았다. Durable subscription 은 실행 중이지 않을 동안에도 Topic 을 Queue 처럼 사용할 수 있다.

7.11 Transaction 의 사용

7.11.1 소개

JEUS JMS 는 local transaction 과 global transaction 모두를 지원한다. 기본적으로, transaction 을 사용하는 session 은 메시지를 보내고 받는 것을 그룹 단위로 처리한다. 그러나 commit 하기 전까지는 message 들을 MessageConsumer 나 MessageListener 에게 전달하지 않는다. 그리고 받은 메시지에 대해서 acknowledge 를 하지 않는다. 만일 rollback 이 된다면 보낸 메시지와 받은 메시지는 모두 무시된다. 이 절에서는 JEUS JMS 에서 어떻게 트랜잭션을 사용하는지에 대해서 설명한다.

7.11.2 Local Transaction 사용

JEUS JMS 내의 로컬 트랜잭션은 글로벌 트랜잭션에 참여하지 않고 JMS session 레벨의 트랜잭션을 의미한다. Session 은 다음 메소드를 사용해서 QueueConnection 또는 TopicConnection 으로부터 트랜잭션을 만든다.

참고: 트랜잭션된 session 을 만들거나 또는 트랜잭션이 되지 않는 session 을 만들든 활성화 중인 transaction 이 존재한다면 session 은 이 활성화된 transaction 에 참여한다. 이 내용과 관련해서는 7.11.3 절을 본다.

```
TopicSession tSes = tCon.createTopicSession(true,0);
QueueSession qSes = qCon.createQueueSession(true,0);
```

두번째 파라미터인 `ack mode` 는 첫번째 파라미터인 `isTransacted` 가 `true` 일 때 무시된다. 클라이언트는 session 이 다음 메소드를 가지고 트랜잭션이 사용되는지 확인할 수 있다.

<<Session.java>>

```
public interface Session extends Runnable {
    ...
    public boolean getTransacted()
        throws JMSException;
```

```
    ...
}
```

Session.commit()이 호출되어야만, 보낸 메시지는 보낸 것으로 간주를 하고, 받은 메시지에 대해서는 acknowledge 한다는 것을 유념하기 바란다. 만일 Session.rollback()이 호출이 되면, 보낸 메시지는 무시되고 받은 메시지는 acknowledge 를 하지 않는다. Session.commit() 뿐만 아니라 Session.rollback()도 불리지 않으면 그것은 rollback 으로 간주한다.

Session.commit 후에 새로운 local session 범위가 자동으로 시작되어야 한다. 그래서 메시지 트랜잭션을 처리하기 위해서 클라이언트 어플리케이션이 새로운 트랜잭션된 session 을 생성하지 않도록 해야 한다.

Session.recover() 메소드와 Message.acknowledge()메소드는 해당 session 이 트랜잭션을 사용할 때 무시된다.

7.11.3 Global Transaction 참여

만일 JEUS JMS session 이 현재 실행 중인 트랜잭션 안에서 만들어졌다면, session 의 isTransacted 타입에 상관없이 'send', 'receive'를 실행하면 트랜잭션에 리소스로 등록된다. 그러나 Session 이 메시지를 보내거나 받는 작업을 하지 않으면, 트랜잭션에 등록이 되지 않는다는 것을 명심하기 바란다.

참고: JEUS 의 JTS 에 관해서 자세히 알고 싶으면, JEUS Server 안내서의 Transaction Manager 부분을 참조하기 바란다.

이런 내용은 EJB 에서도 동일하게 적용이 된다. 예를 들면, Stateless Session Bean 의 메소드 hello 의 트랜잭션 속성값이 Required 값을 가졌다고 가정하자. JMS session 이 이 hello 메소드의 내부에서 만들어지면, 이 JMS session 은 현재 트랜잭션에서 등록이 된다. 이 설명에 대해서 다음 코드를 보자.

<<Hello.java>>

```
package hello;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface Hello extends EJBObject {
    String hello() throws RemoteException;
}
```

<<HelloEJB.java>>

```
package hello;
```

```
    ...
public class HelloEJB implements SessionBean {
    ...
    public void hello()throws RemoteException {
        ...
        tSes = tConn.createTopicSession(
            false,Session.AUTO_ACKNOWLEDGE );
        publisher = tSes.createPublisher(_topic);
        publisher.publish(
            tSes.createTextMessage(“Hello Globe”));
        ...
    }
}
```

tSes 는 현재 트랜잭션에 참여하는 하나의 Session instance 이다.

트랜잭션 컨텍스트는 session 이 생성한 메시지를 통해서는 propagation 이 발생하지 않는다. 그래서 그 메시지의 MessageConsumer 는 트랜잭션에 참여되지 않는다.

Session 이 트랜잭션이 아닐 때 만들어졌고, 현재 트랜잭션이 실행 중 일 때, session 을 생성할 때 사용한 ‘ack mode’나 ‘isTransacted’의 속성은 무시되고 트랜잭션으로 설정이 된다. 다음은 이에 대한 내용을 pseudo-code 로 나타낸 것이다.

```
UserTransaction ut = ejbContext.getUserTransaction();
ut.begin();
    ...
qSes = qConn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE );
sender = qSes.createSender(_queue);
sender.send(qSes.createTextMessage(“Hello Globe”));
    ...
ut.commit();
```

위 예제에서, qSes 의 isTransacted 의 파라미터 '`false`'와 ackmode 파라미터 ‘`Session.AUTO_ACKNOWLEDGE`’는 무시된다. `sender.send()` 메소드가 실행이 될 때, qSes 는 트랜잭션 리소스로 트랜잭션에 등록이 된다.

위 절에서 언급된 JMS session 레벨의 local transaction 과는 별도로, JTA 스펙에서 local transaction 을 제공한다. 그것은 내부적으로 하나의 리소스 매니저

에서만 관리하는 트랜잭션이다. 예를 들어 MessageConsumer 를 통해서 메시지를 받는 session 은 우선 local transaction resource, 정확하게는 local XAResource 로 트랜잭션에 등록하려고 한다. 이렇게 등록이 실패하면, global transaction resource 로 등록된다.

다음 코드는 위의 설정에 대한 예제이다.

```
qSes = qConn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE );
UserTransaction ut = ejbContext.getUserTransaction();
ut.begin();
    ...
sender = qSes.createSender(_queue);
sender.send(qSes.createTextMessage("Hello Globe 1"));
    ...
ut.commit();
sender.send(qSes.createTextMessage("Hello Globe 2"));
```

qSes 는 트랜잭션 범위 안에 만들어지지 않았다. 그러나 메시지를 보내면 트랜잭션에 등록된다. `ut.commit()` 후에도 위의 예제에서는 계속해서 session 을 사용한다. "Hello Globe 2" 메시지를 보낼 때, session 은 `createQueueSession(false, Session.AUTO_ACKNOWLEDGE)`에서 지정한 속성값을 가지고 수행된다. Session 은 동시에 2 개 이상의 트랜잭션에 등록할 수 없다.

7.11.4 Message Driven Bean 의 사용

MDB 에 onMessage 는 메시지가 도착할 때 실행이 된다. 이 메소드는 CMT(Container Managed Transaction)나 BMT(Bean Managed Transaction) 둘 중 하나로 설정할 수 있다. CMT 경우에는 'Required'나 'NotSupported' 둘 중 하나만을 사용해야 한다.

만약 BMT 를 사용한다면, bean 의 onMessage 를 실행시키는 message 수신은 트랜잭션의 일부분이 아니다. 다음 예제를 참조한다.

```
public void onMessage(javax.jms.Message msg){
    ...
    UserTransaction ut =
        ejbContext.getUserTransaction();
    ut.begin();
    ...
    tSes = tConn.createTopicSession(
```

```
        false,Session.AUTO_ACKNOWLEDGE );
    publisher = tSes.createPublisher(_topic);
    javax.jms.Message msg2 =
        tSes.createTextMessage("Hello Globe");
    publisher.publish(msg2);
    ...
    ut.commit();
    ...
}
```

msg 메시지는 ut 에서 begin 한 트랜잭션에 참여하지는 않는다. 그러나 msg2 는 명백히 참여를 한다.

7.11.5 결론

JEUS JMS 는 session 레벨의 local transaction 뿐만 아니라 global transaction 도 지원한다. Session 은 local transaction 으로 메시지를 보내고 받는 것을 그룹으로 할 수 있으며, JTA 서비스에 의해서 제공이 되는 global transaction 내에 참여할 수 있다. JEUS JMS 클라이언트 어플리케이션은 session 을 트랜잭션에 등록하도록 설정할 수 있다.

<<*JMSException.java*>>

```
public class JMSException extends Exception {
    ...
    public String getErrorCode() {
        return this.errorCode;
    }
    ...
}
```

Disconnection 이 발생할 때, JMS 클라이언트는 다음 설정에 따라서 JEUS JMS Server 에 재연결을 시도한다.

- `-Djeus.jms.client.autoReconnect=<true or false>` 기본값은 `true`

만일 파라미터가 true 로 설정이 되어 있다면, 클라이언트 어플리케이션은 JEUS JMS Server 가 재실행이 될 때까지 대기하고, 자동적으로 JEUS JMS Server 에 재연결한다.

7.12 결론

이 절은 JEUS JMS 클라이언트 어플리케이션 개발을 위한 설명의 마지막에 해당 한다. JEUS JMS API 는 클라이언트 어플리케이션이 JMS 1.1 스펙의 프로그램밍 모델에 적합 하도록 하기 위한 설정과 프로그램 작성 방법을 제공 한다.

8 결론

지금까지, JMS 1.1 을 사용해서 메시징 어플리케이션을 개발하는데 필요한 모든 내용을 설명했다. JEUS JMS 의 환경 설정과 JEUS JMS 어플리케이션 개발을 위한 간략한 설명을 하였다.

JEUS JMS 는 JMS 1.1 스펙을 따르고 있으며, JEUS JMS 전용 API 와 파라미터 같은 부가적인 서비스를 제공한다. JEUS JNDI 를 사용하거나, JEUS Transaction Service 에 대한 global 또는 local transaction 참여, 그리고 Servlet, JSP 에서 사용할 수 있고, Message Driven Bean 에서도 사용할 수 있다. JEUS 제품에 통합된 JEUS JMS 는 엔터프라이즈 메시징 어플리케이션 개발에 사용 될 수 있다.

A. jmsadmin Console Tool Reference

A.1 소개

이 부록에는 JEUS JMS Server 에 대한 jmsadmin 을 상세히 살펴본다.

A.2 목적

jmsadmin 은 JEUS JMS Server 를 모니터링하고 관리할 수 있다. JEUS JMS 엔진이 부트가 된 후에, jmsadmin 은 터미널 또는 콘솔창에서 JEUS JMS 엔진을 연결해서 엔진 상태를 체크할 수 있다.

A.3 실행

jmsadmin 은 JEUS_HOME/bin 디렉토리 안에 있다. Windows 플랫폼에서, jmsadmin.bat 로 실행을 하고, 유닉스 플랫폼에서는 jmsadmin 으로 실행한다.

실행은 다음 두 가지 방법이 있다.

```
jmsadmin <node>_containerX
jmsadmin <node>_containerX –Uusername –Ppassword
```

jmsadmin [node]_containerX 명령은 jmsadmin 을 시작하기 전에 부가적인 security check 를 한다. 또한 jmsadmin [node] containerX –UuserName –Ppassword 명령은, 직접적으로 JMS 엔진에 접근할 수 있다.

jmsadmin 으로 연결이 된 상태에서 JEUS 가 down 되면, jmsadmin 콘솔창은 작동하지 않는다. 이 경우에는 다시 엔진을 시작해야 한다.

A.4 명령

다음은 jmsadmin 에 대한 명령들이다.

- **server** : 일반적인 서버 정보를 얻는다.

- **destall** : JMS Server 의 모든 destination 에 대한 정보를 제공한다.

- **dest <destName> | -m <destName> [<message selector>] | –c (–q| -t) <destName> <jndiName> | -r <destName> :** destination 에 관련된 명령이다. destination name 만 지정하면 해당 destination 에 대한 정보가 나오고 –m 을 지정할 경우에는 그 destination 에 저장된 메시지의 정보를 볼수 있다. 이 경우 message selector 를 지정해서 특정 메시지만 확인할수도 있다. –c 를 사용할 경우에는 –q 의 경우 queue, -t 의 경우 topic 을 생성할수 있고 –r 을 사용할 경우에는 해당 destination 을 제거할 수 있다.

- **confall** : JMS Server 의 모든 connection factory 에 대한 정보를 제공한다.

- **conf <factoryName> | –c –q <factoryName> <jndiName> [<max-thread>] | –c –t <factoryName> <jndiName> [<max-thread> [<client-id>]] | -r <factoryName>** : connection factory 에 관련된 명령이다. Factory name 만 제공하면 그 connection factory 의 정보를 볼 수 있다. –c 를 사용하면 connection factory 를 생성할 수 있고 이때 –q 는 queue connection factory, -t 는 topic connection factory 이다. –r 을 사용하면 지정된 connection factory 를 제거할 수 있다.

- **clientall** : JMS Server 에 접속중인 모든 client 에 대한 정보를 제공한다.

- **client <Client Number>** : 해당 client 의 정보를 얻는다.

- **durableall** : JMS Server 에 존재하는 모든 durable subscriber 에 대한 정보를 제공한다.

- **durable <name> [-m [<message selector>]] :** durable subscriber 에 관련된 명령이다. name 만 지정하면 해당 durable subscriber 에 대한 정보가 나오고 –m 을 지정할 경우에는 그 durable subscriber 에 저장된 메시지의 정보를 볼수 있다. 이 경우 message selector 를 지정해서 특정 메시지만 확인할수도 있다.

- **help** or **?** : 콘솔의 도움 정보를 얻는다.

- **exit :** 콘솔에서 빠져 나온다.

B. JMSMain.xml Xml Configuration Reference

B.1 소개

이 부록에서 main conatainer 환경 구성인 JMSMain.xml 의 모든 XML element 를 나열하고 그것들에 대해서 설명한다. Schema 파일은 jms-main.xsds 이고 JEUS_HOME/config/xsds 디렉토리에 위치 한다.

3 개의 하위 절이 있다.

1. XML 트리는, XML 환경구성 파일의 모든 가능한 XML element 들의 완전한 계층적인 리스트이다. 각 element 트리 노드의 형식은 다음과 같다.

    a. 인덱스 숫자는 (예를 들어(11)), 그것은 element 참조에서 세부적인 설명과 관련하여 빠르게 접근 하는데 사용이 된다. 참조 내에 각 element 는 그런 인덱스 숫자에 의해서 행해진다.

    b. XML 의 <element name>은 schema 에 정의가 되어있다.

    c. 기본적인 element 는 XML SCHEMA 에 의해서 정의가 된다. ? 는 없거나 하나의 element 를 가지는 것을 의미하고, +는 하나 또는 그 이상의 elements 들을 가지는 것을 의미하고, *는 하나도 없거나 많은 element 들을 가지는 것을 의미하며, 문자가 없는 것은 정확히 하나의 element 를 가지는 것을 의미한다.

    d. 어떤 element 노드들은 P 문자를 포함하고 있다. 이 element 의 사용과 관련하여 성능의 문제를 의미하는 것을 포함하고 있다. 이것은 환경 구성을 튜닝 하기 위해서 적합하다.

2. **태그 레퍼런스**: 트리에 있는 각 XML 태그를 설명한다.

    a. **Description**: 태그에 대한 간단한 설명

    b. **Value Description**: 입력하는 값과 타입

c. **Value Type**: 값의 데이터 타입. 예: String

d. **Default Value**: 해당 XML 을 사용하지 않았을 때 기본적으로 사용되는 값.

e. **상수**: 이미 정해져 있는 값.

f. **Example**: 해당 XML 태그에 대한 Example.

g. **Performance Recommendation**: 성능 향상을 위해서 추천하는 값.

h. **Child Elements**: 자신의 태그 안에 사용되는 태그.

3. **Example XML 파일**: “JMSMain.xml”에 대한 완전한 예제.

B.2 XML Schema/XML Tree

```
(1) <jms-server>
    (2) <broker-name>?
    (3) <service-config>+
        (4) <name>
        (5) <server-address>? P
        (6) <port>
        (7) <use-ssl>? P
        (8) <client-limit>? P
        (9) <backlog-size>? P
        (10) <blocking-socket>? P
        (11) <read-buffer-size>? P
        (12) <write-buffer-size>? P
        (13) <connection-timeout>? P
        (14) <check-security>? P
        (15) <client-keepalive-timeout>? P
    (16) <thread-pool>?
        (17) <min>? P
        (18) <max>? P
        (19) <keep-alive-time>? P
    (20) <access-log>?
        (21) <level>? P
        (22) <use-parent-handlers>? P
```

```
(23) <filter-class>?
(24) <handler>?
    (25) <console-handler>
        (26) <name>
        (27) <level>? P
        (28) <encoding>?
        (29) <filter-class>?
    (30) <file-handler>
        (31) <name>
        (32) <level>? P
        (33) <encoding>?
        (34) <filter-class>?
        (35) <file-name>?
        (36) <valid-day>
        (37) <valid-hour>
        (38) <buffer-size>? P
        (39) <append>? P
    (40) <smtp-handler>
        (41) <name>
        (42) <level>? P
        (43) <encoding>?
        (44) <filter-class>?
        (45) <smtp-host-address>
        (46) <from-address>
        (47) <to-address>
        (48) <cc-address>?
        (49) <bcc-address>?
        (50) <send-for-all-messages>? P
    (51) <socket-handler>
        (52) <name>
        (53) <level>? P
        (54) <encoding>?
        (55) <filter-class>?
        (56) <address>
        (57) <port>
    (58) <user-handler>
        (59) <handler-class>
        (60) <name>
        (61) <level>? P
```

```
                (62) <encoding>?
                (63) <filter-class>?
                (64) <handler-property>?
                    (65) <property>*
                        (66) <key>
                        (67) <value>
                (68) <formatter-class>? P
                (69) <formatter-property>?
                    (70) <property>*
                        (71) <key>
                        (72) <value>
(73) <durable-subscriber>*
    (74) <client-id>
    (75) <name>
    (76) <destination-name>
    (77) <message-selector>?
(78) <connection-factory>*
    (79) <type>? P
    (80) <name>
    (81) <service>?
    (82) <export-name>?
    (83) <fixed-client-id>? P
    (84) <client-id>?
    (85) <thread-pool>?
        (86) <min>? P
        (87) <max>? P
        (88) <keep-alive-time>? P
(89) <destination>*
    (90) <type>
    (91) <name>
    (92) <export-name>?
    (93) <multiple-receiver>? P
    (94) <local-distribute>? P
    (95) <limit>? P
    (96) <high-mark>? P
    (97) <low-mark>? P
    (98) <queue-cluster-name>?
    (99) <topic-cluster-name>?
(100) <queue-cluster>*
```

```
    (101) <queue-cluster-name>
    (102) <cluster-distribute>? P
    (103) <local-preference>? P
    (104) <target>+
(105) <topic-cluster>*
    (106) <topic-cluster-name>
    (107) <relay-type> P
    (108) <target>+
(109) <storage>?
    (110) <db-storage>?
        (111) <data-source-name>
        (112) <vendor>?
        (113) <destination-table-name>? P
        (114) <message-table-name>? P
        (115) <relation-table-name>? P
        (116) <subscription-table-name>? P
        (117) <transaction-table-name>? P
        (118) <key-table-name>? P
    (119) <file-storage>?
        (120) <path>
        (121) <file-access-mode>? P
        (122) <message-file>?
            (123) <initial-file-size>? P
            (124) <limit-file-size>? P
            (125) <block-size>? P
            (126) <increase-rate>? P
(127) <jms-server-cluster>?
    (128) <cluster-service>?
    (129) <jms-server-info>+
        (130) <broker-name>
        (131) <broker-id>
        (132) <ip-address>
        (133) <jms-port>?
        (134) <use-ssl>? P
        (135) <connection-interval>? P
        (136) <weight>? P
```

B.3 Element Reference

(1) **<jms-server>**

Description JMSMain.xml 내의 최고 레벨의 element 이다. 내부 설정값들 중 token 혹은 String 에 해당하는 값들은 다음의 문자들을 포함할 수 없다. ':', ',', '*', '?', '='

Child Elements
(2)broker-name?
(3)service-config+
(16)thread-pool?
(20)access-log?
(73)durable-subscriber*
(78)connection-factory*
(89)destination*
(100)queue-cluster*
(105)topic-cluster*
(109)storage?
(127)jms-server-cluster?

(2) <jms-server> **<broker-name>**

Description JMS 메시지 브로커의 이름이다. 설정하지 않는 경우 호스트 이름이 사용된다.

Value Type token

(3) <jms-server> **<service-config>**

Description 메시징 서비스를 제공하기 위한 서버 채널에 대한 설정이다. 최소 하나 이상 설정되어야 한다.

Child Elements
(4)name
(5)server-address?
(6)port
(7)use-ssl?
(8)client-limit?
(9)backlog-size?
(10)blocking-socket?
(11)read-buffer-size?
(12)write-buffer-size?
(13)connection-timeout?
(14)check-security?
(15)client-keepalive-timeout?

(4) <jms-server> <service-config> **<name>**

Description 서버 채널의 이름이다. Connection Factory 에 채널 정보를 지정하기 위해

용도로 사용된다.

Value Type token

(5) <jms-server> <service-config> **<server-address>**

Description 해당 서버 채널의 IP 어드레스를 직접 설정한다. 클라이언트가 접속하기 위한 주소값으로 사용된다. 설정하지 않는 경우 "127.0.0.1" 이 사용된다.

Value Type token

Default Value 127.0.0.1

Example `123.123.123.123`

(6) <jms-server> <service-config> **<port>**

Description 해당 서버 채널의 서비스 포트 번호 이다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Example `9741`

(7) <jms-server> <service-config> **<use-ssl>**

Description 서버 채널에 SSL 을 적용할 것인지를 설정한다.

Value Type boolean

Default Value false

(8) <jms-server> <service-config> **<client-limit>**

Description 서버 채널이 허용하는 최대 클라이언트 수를 지정한다. blocking socket 을 사용하는 경우 메시지 브로커에 설정된 최대 쓰레드 수를 넘을 수 없다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 1000

(9) <jms-server> <service-config> **<backlog-size>**

Description 서버 채널이 허용하는 백로그의 크기이다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value	100

(10) <jms-server> <service-config> **<blocking-socket>**

Description	해당 서버 채널이 blocking socket 을 사용할 것인가 non-blocking socket 을 사용할 것인가를 결정한다. 확장성을 위해 non-blocking socket 을 사용할 것을 권장한다.
Value Type	boolean
Default Value	true

(11) <jms-server> <service-config> **<read-buffer-size>**

Description	클라이언트로 부터 메시지를 읽어 들이는 네트워크 버퍼의 크기이다. 디폴트는 64K 이다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	65536

(12) <jms-server> <service-config> **<write-buffer-size>**

Description	클라이언트에게 메시지를 전송하기 위한 네트워크 버퍼의 크기이다. 디폴트는 64K 이다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	65536

(13) <jms-server> <service-config> **<connection-timeout>**

Description	blocking socket 을 사용할 경우 지정된 시간동안 클라이언트로 부터 아무런 요청이 없으면 연결을 종료한다. 분단위로 설정하며 디폴트는 120 분 이다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	120

(14) <jms-server> <service-config> **<check-security>**

Description	Connection Factory 에서 connection 을 만들때 security check 를 할지의 여부를 결정한다. 이 element 의 값이 true 일때 user 는 jeus.jms.client.connectionFactory permission 을 가지고 있어야 하고 createConnection action 이 허용되어야 한다.

Value Type boolean

Default Value false

(15) <jms-server> <service-config> **<client-keepalive-timeout>**

Description 클라이언트와의 연결이 비정상 종료되었을 경우 재접속을 기다리는 시간이다. 이 시간이 지나면 해당 클라이언트의 리소스는 모두 서버에 반환된다. 설정된 시간내에는 해당 클라이언트의 clientID 값이 유지되므로 네트워크 상태가 불량한 경우에만 설정하도록 한다. 초단위로 설정하며 0 이하의 값을 지정하면 기다리지 않고 즉시 리소스를 반환한다.

Value Type int

Default Value 30

(16) <jms-server> **<thread-pool>**

Description JMS 메시지 브로커가 사용하는 thread pool 에 대한 설정이다.

Child Elements
(17)min?
(18)max?
(19)keep-alive-time?

(17) <jms-server> <thread-pool> **<min>**

Description thread pool 의 최소 크기를 지정한다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 0

(18) <jms-server> <thread-pool> **<max>**

Description thread pool 의 최대 크기를 지정한다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 300

(19) <jms-server> <thread-pool> **<keep-alive-time>**

Description min 갯수를 초과하는 thread 에 대해서 여기에 지정된 시간동안 사용되지 않은 thread 는 소멸하게 된다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 300000

(20) <jms-server> **<access-log>**

Description access log 에 대한 정보를 지정한다.

Child Elements
(21)level?
(22)use-parent-handlers?
(23)filter-class?
(24)handler?

(21) <jms-server> <access-log> **<level>**

Description logging 의 level 을 설정한다. 각 level 의 의미는 J2SE 의 logging API 의 Level class 를 참고하기 바란다.

Value Type loggingLevelType

Default Value INFO

Defined Value

FATAL
SEVERE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

NOTICE
WARNING 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

INFORMATION
INFO 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

DEBUG
FINE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

SEVERE
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

WARNING
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

INFO
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

CONFIG

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINE

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINER

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINEST

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

ALL

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

(22) <jms-server> <access-log> **<use-parent-handlers>**

Description parent logger 의 handler 들을 이 logger 에서도 사용할지를 결정한다.

Value Type boolean

Default Value false

(23) <jms-server> <access-log> **<filter-class>**

Description logger 에 지정할 filer class 의 fully qualified class name 을 설정한다.

Value Type token

Example

```
<filter-class>com.tmax.logging.filter.MyFilter</filter-class>
```

(24) <jms-server> <access-log> **<handler>**

Description logger 에서 사용할 handler 를 설정한다.

Child Elements

```
(25)console-handler
(30)file-handler
(40)smtp-handler
(51)socket-handler
(58)user-handler
```

(25) <jms-server> <access-log> <handler> **<console-handler>**

Description logging 을 화면에 남기고자 하는 경우에 사용하는 handler 이다.

Child Elements

```
(26)name
```

```
(27)level?
(28)encoding?
(29)filter-class?
```

(26) <jms-server> <access-log> <handler> <console-handler> <name>

Description handler 의 이름을 설정한다. 이때 이 이름은 하나의 logger 내에서만 unique 하게 설정되면 된다. 이 이름은 tool 등에서 handler 를 지칭할 때 사용한다.

Value Type token

Example
```
<name>handler1</name>
```

(27) <jms-server> <access-log> <handler> <console-handler> <level>

Description 이 handler 의 level 을 설정한다. logger 를 통과한 message 의 level 이 이 handler level 에 포함될 경우에만 이 handler 에 의해 출력된다.

Value Type loggingLevelType

Default Value FINEST

Defined Value

FATAL
SEVERE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

NOTICE
WARNING 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

INFORMATION
INFO 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

DEBUG
FINE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

SEVERE
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

WARNING
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

INFO

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

CONFIG

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINE

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINER

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINEST

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

ALL

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

(28) <jms-server> <access-log> <handler> <console-handler> **<encoding>**

Description 이 handler 이 message 를 남길때 사용할 encoding 을 설정한다.

Value Type token

(29) <jms-server> <access-log> <handler> <console-handler> **<filter-class>**

Description 이 handler 에 지정할 filer class 의 fully qualified class name 을 설정한다.

Value Type token

Example

```
<filter-class>com.tmax.logging.filter.MyFilter</filter-class>
```

(30) <jms-server> <access-log> <handler> **<file-handler>**

Description logging 을 file 로 출력하고자 하는 경우에 사용하는 handler 이다.

Child Elements

```
(31)name
(32)level?
(33)encoding?
(34)filter-class?
(35)file-name?
(36)valid-day
(37)valid-hour
(38)buffer-size?
(39)append?
```

(31) <jms-server> <access-log> <handler> <file-handler> **<name>**

Description handler 의 이름을 설정한다. 이때 이 이름은 하나의 logger 내에서만 unique 하게 설정되면 된다. 이 이름은 tool 등에서 handler 를 지칭할 때 사용한다.

Value Type token

Example
```
<name>handler1</name>
```

(32) <jms-server> <access-log> <handler> <file-handler> **<level>**

Description 이 handler 의 level 을 설정한다. logger 를 통과한 message 의 level 이 이 handler level 에 포함될 경우에만 이 handler 에 의해 출력된다.

Value Type loggingLevelType

Default Value FINEST

Defined Value

FATAL
SEVERE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

NOTICE
WARNING 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

INFORMATION
INFO 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

DEBUG
FINE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

SEVERE
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

WARNING
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

INFO
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

CONFIG
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINE
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINER
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINEST
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

ALL
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

(33) <jms-server> <access-log> <handler> <file-handler> **<encoding>**

Description 이 handler 이 message 를 남길때 사용할 encoding 을 설정한다.

Value Type token

(34) <jms-server> <access-log> <handler> <file-handler> **<filter-class>**

Description 이 handler 에 지정할 filer class 의 fully qualified class name 을 설정한다.

Value Type token

Example
```
<filter-class>com.tmax.logging.filter.MyFilter</filter-class>
```

(35) <jms-server> <access-log> <handler> <file-handler> **<file-name>**

Description 이 handler 가 사용할 file name 을 설정한다. 만약 user 가 이 설정을 하지 않으면 각 logger 의 default file name 이 사용된다. 각각의 default file name 은 JEUS Server 매뉴얼을 참고하기 바란다.

Value Type token

Example
```
<file-name>C:\logs\mylog.log</file-name>
```

(36) <jms-server> <access-log> <handler> <file-handler> **<valid-day>**

Description 이 handler 가 사용하는 file 을 이 설정에서 지정된 기간동안만 사용하고 계속 갱신할 경우에 사용한다. 이 설정은 날짜 단위로 file 을 바꿀때 사용한다. 이 경우 handler 가 사용하는 file 이름 뒤에 file 이 사용된 날짜가 자동으로 붙게 된다.

Value Description day

Value Type off-intType

Value Type Description 기본적으로 non-negative int 형이지만 -1 인 경우에는 설정을 하지 않은게 된다.즉, off 된다.

Example `<valid-day>1</valid-day>`

(37) <jms-server> <access-log> <handler> <file-handler> **<valid-hour>**

Description 이 handler 가 사용하는 file 을 이 설정에서 지정된 기간동안만 사용하고 계속 갱신할 경우에 사용한다. 이 설정은 시간 단위로 file 을 바꿀때 사용한다. 이 경우 handler 가 사용하는 file 이름 뒤에 file 이 사용된 날짜와 시간이 자동으로 붙게 된다.

Value Description 시간을 나타내며 24 의 약수 + n*24 (n 은 0 이상의 정수)의 값을 가져야 한다.

Value Type off-intType

Value Type Description 기본적으로 non-negative int 형이지만 -1 인 경우에는 설정을 하지 않은게 된다.즉, off 된다.

Example `<valid-hour>3</valid-hour>`

(38) <jms-server> <access-log> <handler> <file-handler> **<buffer-size>**

Description 이 handler 가 file 에 출력할때 사용하는 buffer 의 크기를 지정한다.

Value Description byte 단위이다. [Performance Recommandation]: 이 값이 클수록 file 에 출력되는 message 는 지연되어 출력되지만 logging 성능은 좋아진다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 1024

(39) <jms-server> <access-log> <handler> <file-handler> **<append>**

Description 이 handler 가 사용하는 file 이 이미 존재하는 경우 file 뒤에 덧붙여 쓸지를 결정한다. false 로 설정되어 있다면 기존의 file 은 제거된다.

Value Type boolean

Default Value true

(40) <jms-server> <access-log> <handler> **<smtp-handler>**

Description logging 을 email 로 보내고자 하는 경우에 사용하는 handler 이다. [Performance Recommandation]: logging message 하나가 하나의 email 로 전송되므로 적절한 filter 없이 사용하는 것은 엄청난 양의 email 을 발생시켜 아주 위험하므로 주의를 요한다.

Child Elements

(41)name
(42)level?
(43)encoding?
(44)filter-class?
(45)smtp-host-address
(46)from-address
(47)to-address
(48)cc-address?
(49)bcc-address?
(50)send-for-all-messages?

(41) <jms-server> <access-log> <handler> <smtp-handler> **<name>**

Description

handler 의 이름을 설정한다. 이때 이 이름은 하나의 logger 내에서만 unique 하게 설정되면 된다. 이 이름은 tool 등에서 handler 를 지칭할 때 사용한다.

Value Type

token

Example

```
<name>handler1</name>
```

(42) <jms-server> <access-log> <handler> <smtp-handler> **<level>**

Description

이 handler 의 level 을 설정한다. logger 를 통과한 message 의 level 이 이 handler level 에 포함될 경우에만 이 handler 에 의해 출력된다.

Value Type

loggingLevelType

Default Value

FINEST

Defined Value

FATAL
SEVERE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

NOTICE
WARNING 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

INFORMATION
INFO 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

DEBUG
FINE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

SEVERE

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

WARNING

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

INFO

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

CONFIG

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINE

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINER

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINEST

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

ALL

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

(43) <jms-server> <access-log> <handler> <smtp-handler> **<encoding>**

Description 이 handler 이 message 를 남길때 사용할 encoding 을 설정한다.

Value Type token

(44) <jms-server> <access-log> <handler> <smtp-handler> **<filter-class>**

Description 이 handler 에 지정할 filer class 의 fully qualified class name 을 설정한다.

Value Type token

Example

```
<filter-class>com.tmax.logging.filter.MyFilter</filter-class>
```

(45) <jms-server> <access-log> <handler> <smtp-handler> **<smtp-host-address>**

Description email 을 보낼 smtp server 의 주소를 지정한다.

Value Type token

(46) <jms-server> <access-log> <handler> <smtp-handler> **<from-address>**

Description email 을 보내는 사람의 address 를 지정한다.

Value Type token

(47) <jms-server> <access-log> <handler> <smtp-handler> **<to-address>**

Description email 을 받는 사람의 address 를 지정한다.

Value Type token

(48) <jms-server> <access-log> <handler> <smtp-handler> **<cc-address>**

Description email 을 참조로 받는 사람의 address 를 지정한다.

Value Type token

(49) <jms-server> <access-log> <handler> <smtp-handler> **<bcc-address>**

Description email 을 숨은 참조로 받는 사람의 address 를 지정한다.

Value Type token

(50) <jms-server> <access-log> <handler> <smtp-handler> **<send-for-all-messages>**

Description 이 handler 가 등록한 logger 의 log() method 를 통해 들어온 message 들이 이 handler 로 들어왔을때 이를 email 로 보낼 대상으로 여길지를 설정한다. 만약 false 로 설정되어 있으면 logger 의 특별한 send() method 로 호출된 message 들만 email 로 전송된다. 즉, 처음부터 email 로 보낼 의도로 지정된 message 들만 email 로 전송된다.

Value Type boolean

Default Value false

(51) <jms-server> <access-log> <handler> **<socket-handler>**

Description logging 을 지정된 socket 으로 보내고자 하는 경우에 사용하는 handler 이다. [Performance Recommandation]: logging message 하나당 Socket 으로 전송이 되므로 적절한 filter 없이 사용하는 것은 성능 저하를 가져온다.

Child Elements
(52)name
(53)level?
(54)encoding?
(55)filter-class?
(56)address
(57)port

(52) <jms-server> <access-log> <handler> <socket-handler> **<name>**

Description handler 의 이름을 설정한다. 이때 이 이름은 하나의 logger 내에서만 unique 하게 설정되면 된다. 이 이름은 tool 등에서 handler 를 지칭할 때 사용한다.

Value Type token

Example
```
<name>handler1</name>
```

(53) <jms-server> <access-log> <handler> <socket-handler> **<level>**

Description 이 handler 의 level 을 설정한다. logger 를 통과한 message 의 level 이 이 handler level 에 포함될 경우에만 이 handler 에 의해 출력된다.

Value Type loggingLevelType

Default Value FINEST

Defined Value

FATAL
SEVERE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

NOTICE
WARNING 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

INFORMATION
INFO 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

DEBUG
FINE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

SEVERE
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

WARNING
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

INFO
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

CONFIG
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINE

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINER

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINEST

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

ALL

J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

(54) <jms-server> <access-log> <handler> <socket-handler> **<encoding>**

Description 이 handler 이 message 를 남길때 사용할 encoding 을 설정한다.

Value Type token

(55) <jms-server> <access-log> <handler> <socket-handler> **<filter-class>**

Description 이 handler 에 지정할 filer class 의 fully qualified class name 을 설정한다.

Value Type token

Example

```
<filter-class>com.tmax.logging.filter.MyFilter</filter-class>
```

(56) <jms-server> <access-log> <handler> <socket-handler> **<address>**

Description 이 handler 가 생성될때 message 들을 보낼 곳의 IP address 를 설정한다.

Value Type token

(57) <jms-server> <access-log> <handler> <socket-handler> **<port>**

Description 이 handler 가 생성될때 message 들을 보낼 곳의 port 를 설정한다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

(58) <jms-server> <access-log> <handler> **<user-handler>**

Description User 가 J2SE logging API 에 따라 만든 handler 를 사용할 경우의 설정이다.

Child Elements

```
(59)handler-class
(60)name
(61)level?
```

(62)encoding?
(63)filter-class?
(64)handler-property?
(68)formatter-class?
(69)formatter-property?

(59) <jms-server> <access-log> <handler> <user-handler> **<handler-class>**

Description user 가 만든 handler 의 fully qualified class name 을 설정한다.이 클래스는 java.util.logging.Handler 를 상속받고 jeus.util.logging.JeusHandler 를 구현해야 한다.

Value Type token

Example
```
<handler-
class>com.tmax.logging.handler.MyHandler</handler-
class>
```

(60) <jms-server> <access-log> <handler> <user-handler> **<name>**

Description handler 의 이름을 설정한다. 이때 이 이름은 하나의 logger 내에서만 unique 하게 설정되면 된다. 이 이름은 tool 등에서 handler 를 지칭할 때 사용한다.

Value Type token

Example
```
<name>handler1</name>
```

(61) <jms-server> <access-log> <handler> <user-handler> **<level>**

Description 이 handler 의 level 을 설정한다. logger 를 통과한 message 의 level 이 이 handler level 에 포함될 경우에만 이 handler 에 의해 출력된다.

Value Type loggingLevelType

Default Value FINEST

Defined Value
FATAL
SEVERE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

NOTICE
WARNING 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

INFORMATION
INFO 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

DEBUG
FINE 에 해당하는 level 이다. JEUS 4.x 대의 logger 의 level 로 compatibility 를 위해 지원한다.

SEVERE
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

WARNING
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

INFO
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

CONFIG
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINE
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINER
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

FINEST
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

ALL
J2SE Logging API 의 Level class documentation 을 참고하기 바란다.

(62) <jms-server> <access-log> <handler> <user-handler> **<encoding>**

Description 이 handler 이 message 를 남길때 사용할 encoding 을 설정한다.

Value Type token

(63) <jms-server> <access-log> <handler> <user-handler> **<filter-class>**

Description 이 handler 에 지정할 filer class 의 fully qualified class name 을 설정한다.

Value Type token

Example
```
<filter-class>com.tmax.logging.filter.MyFilter</filter-class>
```

(64) <jms-server> <access-log> <handler> <user-handler> **<handler-property>**

Description handler 가 생성될 때 넘겨줄 property 를 설정한다. 이 property 들은 key-value 로 Map 객체에 저장되어 JeusHandler.setProperty() method 를 통해 handler 로 전달된다.

Child Elements (65)property*

(65) <jms-server> <access-log> <handler> <user-handler> <handler-property> **<property>**

Description handler 등에게 전달할 property 들을 설정한다.

Child Elements (66)key
(67)value

(66) <jms-server> <access-log> <handler> <user-handler> <handler-property> <property> **<key>**

Description property 의 key 값이다.

Value Type token

(67) <jms-server> <access-log> <handler> <user-handler> <handler-property> <property> **<value>**

Description property 의 value 값이다.

Value Type token

(68) <jms-server> <access-log> <handler> <user-handler> **<formatter-class>**

Description 이 handler 가 사용할 formatter 의 fully qualified class name 을 설정한다.이 클래스는 java.util.logging.Formatter 를 상속받고 jeus.util.logging.JeusFormatter 를 구현해야 한다.

Value Type token

Default Value jeus.util.logging.SimpleFormatter

Example
```
<formatter-class>com.tmax.logging.handler.MyHandler</formatter-class>
```

(69) <jms-server> <access-log> <handler> <user-handler> **<formatter-property>**

Description handler 가 생성될 때 만들어진 formatter 에게 넘겨줄 property 를 설정한다. 이 property 들은 key-value 로 Map 객체에 저장되어 JeusFormatter.setProperty() method 를 통해 formatter 로 전달된다.

Child Elements (70)property*

(70) <jms-server> <access-log> <handler> <user-handler> <formatter-property> **<property>**

Description handler 등에게 전달할 property 들을 설정한다.

Child Elements
(71)key
(72)value

(71) <jms-server> <access-log> <handler> <user-handler> <formatter-property> <property> **<key>**

Description property 의 key 값이다.

Value Type token

(72) <jms-server> <access-log> <handler> <user-handler> <formatter-property> <property> **<value>**

Description property 의 value 값이다.

Value Type token

(73) <jms-server> **<durable-subscriber>**

Description Durable subscriber 에 대한 정보를 지정한다.

Child Elements
(74)client-id
(75)name
(76)destination-name
(77)message-selector?

(74) <jms-server> <durable-subscriber> **<client-id>**

Description 이 태그는 클라이언트를 식별하는 값을 설정한다. connection-factory element 내에서 뿐만 아니라 durable-subscirber element 내에서 모든 client-id 값들 중에서 중복 되어서는 안된다.

Value Type token

(75) <jms-server> <durable-subscriber> **<name>**

Description JMS 메시지 브로커 내에서 관리 목적으로 사용되는 Durable Subscriber 의 이름이다.

Value Type token

(76) <jms-server> <durable-subscriber> **<destination-name>**

Description Durable Subscriber 가 메시지를 받고자 하는 데스티네이션의 이름이다.

Value Type token

(77) <jms-server> <durable-subscriber> **<message-selector>**

Description Durable Subscriber 의 message selector 를 설정한다.

Value Type token

(78) <jms-server> **<connection-factory>**

Description connection factory 에 대한 정보를 지정한다.

Child Elements
(79)type?
(80)name
(81)service?
(82)export-name?
(83)fixed-client-id?
(84)client-id?
(85)thread-pool?

(79) <jms-server> <connection-factory> **<type>**

Description 해당 Connection Factory 의 종류를 설정한다.

Value Type factory-typeType

Default Value nonxa

Defined Value nonxa

xa

queue

topic

xaqueue

xatopic

(80) <jms-server> <connection-factory> **<name>**

Description JMS 시스템 내에서 관리의 목적으로 사용되는 Connection Factory 의 이름이다.

Value Type token

(81) <jms-server> <connection-factory> **<service>**

Description 해당 Connection Factory 가 연결을 시도할 service 의 이름을 설정한다. service-config 에 지정된 이름을 사용하도록 한다. 이름을 지정하지 않을 경우 처음 설정된 서비스로 임의 지정된다.

Value Type token

(82) <jms-server> <connection-factory> **<export-name>**

Description 해당 Connection Factory 가 네이밍 서버에 binding 되는 이름. 설정하지 않으면 name 속성이 그대로 사용된다.

Value Type token

(83) <jms-server> <connection-factory> **<fixed-client-id>**

Description connection ID 의 생성 정책이다. true 일 경우 지정된 client-id 를 그대로 사용하며 false 인 경우 JMS 메시지 브로커에 의해 자동적으로 생성된다. 생성된 connection ID 는 getClientID() API 를 이용하여 확인할 수 있다.

Value Type boolean

Default Value true

(84) <jms-server> <connection-factory> **<client-id>**

Description 해당 Connection Factory 를 이용하여 생성되는 connection 에 기본값으로 설정되는 ClientID 값이다.

Value Type token

(85) <jms-server> <connection-factory> **<thread-pool>**

Description client 가 이 Connection Factory 로 부터 connection 을 만들 경우 그 connection 에 대한 thread pool 의 정보를 지정한다.

Child Elements
(86)min?
(87)max?
(88)keep-alive-time?

(86) <jms-server> <connection-factory> <thread-pool> **<min>**

Description thread pool 의 최소 크기를 지정한다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 0

(87) <jms-server> <connection-factory> <thread-pool> **<max>**

Description thread pool 의 최대 크기를 지정한다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 300

(88) <jms-server> <connection-factory> <thread-pool> **<keep-alive-time>**

Description min 갯수를 초과하는 thread 에 대해서 여기에 지정된 시간동안 사용되지 않은 thread 는 소멸하게 된다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 300000

(89) <jms-server> **<destination>**

Description 데스티네이션에 대한 정보를 지정한다.

Child Elements
(90)type
(91)name
(92)export-name?
(93)multiple-receiver?
(94)local-distribute?
(95)limit?
(96)high-mark?
(97)low-mark?
(98)queue-cluster-name?
(99)topic-cluster-name?

(90) <jms-server> <destination> **<type>**

Description 해당 데스티네이션의 종류를 설정한다. queue 와 topic 중 하나를 지정한다.

Value Type destination-typeType

Defined Value queue

topic

(91) <jms-server> <destination> **<name>**

Description JMS 메시지 브로커 내에서 관리 목적으로 사용되는 데스티네이션의 이름이다.

Value Type token

(92) <jms-server> <destination> **<export-name>**

Description 해당 데스티네이션이 네이밍 서버에 binding 되는 이름. 설정하지 않으면 name 속성이 그대로 사용된다.

Value Type token

(93) <jms-server> <destination> **<multiple-receiver>**

Description queue 타입의 데스티네이션에 대해 다중의 receiver 를 허용하여 분산처리를 할 것인지를 설정한다.

Value Type boolean

Default Value false

(94) <jms-server> <destination> **<local-distribute>**

Description multiple-receiver 로 설정된 queue 데스티네이션의 메시지 분산 방식을 결정한다. round-robin, random 중에 하나를 지정할 수 있다.

Value Type local-distributeType

Default Value round-robin

Defined Value round-robin

random

(95) <jms-server> <destination> **<limit>**

Description 해당 destination 에서 사용할수 있는 최대 메모리 크기를 설정한다. 사용중인 메모리가 이 값을 초과하는 경우 클라이언트의 메시지 전달은 바로 에러 처리되게 된다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 2147483647

(96) <jms-server> <destination> **<high-mark>**

Description 플로우 컨트롤을 사용하기 시작하는 메모리 크기를 설정한다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 2147483647

(97) <jms-server> <destination> **<low-mark>**

Description 소프트 캐싱을 시작하는 메모리 크기를 설정한다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 2147483647

(98) <jms-server> <destination> **<queue-cluster-name>**

Description queue-cluster 로 지정된 클러스터링 타입을 선택한다.

Value Type token

(99) <jms-server> <destination> **<topic-cluster-name>**

Description topic-cluster 로 지정된 클러스터링 타입을 선택한다.

Value Type token

(100) <jms-server> **<queue-cluster>**

Description queue 클러스터링 방식을 설정한다.

Child Elements
(101)queue-cluster-name
(102)cluster-distribute?
(103)local-preference?
(104)target+

(101) <jms-server> <queue-cluster> **<queue-cluster-name>**

Description queue 클러스터링을 위한 클러스터링 타입을 설정한다.

Value Type token

(102) <jms-server> <queue-cluster> **<cluster-distribute>**

Description queue 클러스터링에 참여하는 각각의 브로커에 메시지를 분산하기 위한 방식을 지정한다. server-weighted, consumer-weighted, round-robin, random 중에 하나를 지정할 수 있다.

Value Type cluster-distributeType

Default Value round-robin

Defined Value server-weighted

consumer-weighted

round-robin

random

(103) <jms-server> <queue-cluster> **<local-preference>**

Description local 브로커에서 처리하는 메시지의 비중을 높인다. server-weighted 혹은 consumer-weighted 방식을 사용할 경우에만 의미가 있다.

Value Type positiveIntType

Value Type Description 기본적으로 non-negative int 형이지만 1 이상의 값만 허용한다.

Default Value 1

(104) <jms-server> <queue-cluster> **<target>**

Description 클러스터링에 참여하는 broker name 을 지정한다. 해당 이름은 jms-server-info 에 설정된 이름이어야 한다.

Value Type token

(105) <jms-server> **<topic-cluster>**

Description topic 클러스터링 방식을 설정한다.

Child Elements
(106)topic-cluster-name
(107)relay-type
(108)target+

(106) <jms-server> <topic-cluster> **<topic-cluster-name>**

Description topic 클러스터링을 위한 클러스터링 타입을 설정한다.

Value Type token

(107) <jms-server> <topic-cluster> **<relay-type>**

Description 위에서 지정된 cluster 서버에 message 를 전달하는 방식이다. non-durable 과 durable 을 지정할 수 있다.

Value Type relay-typeType

Default Value non-durable

Defined Value non-durable

durable

(108) <jms-server> <topic-cluster> **<target>**

Description 클러스터링에 참여하는 broker name 을 지정한다. 해당 이름은 jms-server-info 에 설정된 이름이어야 한다.

Value Type token

(109) <jms-server> **<storage>**

Description persistent message 를 위한 storage 정보를 설정한다.

Child Elements
(110)db-storage?
(119)file-storage?

(110) <jms-server> <storage> **<db-storage>**

Description database storage 를 사용하는 경우 database storage 에 대한 설정을 한다.

Child Elements
(111)data-source-name
(112)vendor?
(113)destination-table-name?
(114)message-table-name?
(115)relation-table-name?
(116)subscription-table-name?
(117)transaction-table-name?
(118)key-table-name?

(111) <jms-server> <storage> <db-storage> **<data-source-name>**

Description 사용하고자 하는 JNDI 에 binding 된 datasource 의 이름을 지정한다.

Value Type token

(112) <jms-server> <storage> <db-storage> **<vendor>**

Description 해당 DataSource 가 연결된 DB 종류를 명시한다. 지정되어 있지 않으면 JEUSMain.xml 에 등록된 해당 DataSource 의 vendor 가 사용된다.

Value Type token

(113) <jms-server> <storage> <db-storage> **<destination-table-name>**

Description JMS 에서 사용하는 destination table name 을 지정한다.

Value Type token

Default Value jms_dest

(114) <jms-server> <storage> <db-storage> **<message-table-name>**

Description JMS 에서 사용하는 message table name 을 지정한다.

Value Type token

Default Value jms_message

(115) <jms-server> <storage> <db-storage> **<relation-table-name>**

Description JMS 에서 사용하는 relation table name 을 지정한다.

Value Type token

Default Value jms_relation

(116) <jms-server> <storage> <db-storage> **<subscription-table-name>**

Description JMS 에서 사용하는 subscription table name 을 지정한다.

Value Type token

Default Value jms_sub

(117) <jms-server> <storage> <db-storage> **<transaction-table-name>**

Description JMS 에서 사용하는 transaction table name 을 지정한다.

Value Type token

Default Value jms_tx

(118) <jms-server> <storage> <db-storage> **<key-table-name>**

Description JMS 에서 사용하는 key table name 을 지정한다.

Value Type token

Default Value jms_key

(119) <jms-server> <storage> **<file-storage>**

Description file storage 를 사용하는 경우 file storage 에 대한 설정을 한다.

Child Elements
(120)path
(121)file-access-mode?
(122)message-file?

(120) <jms-server> <storage> <file-storage> **<path>**

Description file storage 가 위치하는 full file path 이다.

Value Type token

(121) <jms-server> <storage> <file-storage> **<file-access-mode>**

Description file storage 에 적용되는 access mode 이며 처리되는 데이터의 중요도에 따라 설정하도록 한다. cache 또는 sync 를 설정할 수 있으며 sync 의 경우 저장 매체에 상당한 부하를 줄 수 있다. cache 를 설정하는 경우 비정상적인 시스템 셧다운시 일부 메시지가 유실될 수도 있다.

Value Type access-modeType

Default Value cache

Defined Value cache

sync

(122) <jms-server> <storage> <file-storage> **<message-file>**

Description message 가 저장되는 파일에 대한 설정이다.

Child Elements
(123)initial-file-size?
(124)limit-file-size?
(125)block-size?
(126)increase-rate?

(123) <jms-server> <storage> <file-storage> <message-file> **<initial-file-size>**

Description file storage 최초 생성시의 크기.디폴트는 4M 이다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 4194304

(124) <jms-server> <storage> <file-storage> <message-file> **<limit-file-size>**

Description file storage 의 최대 크기. 디폴트는 40G 이다.

Value Type nonNegativeLongType

Value Type Description 0 이상의 long type 이다. 즉, long 범위에서 0 이상의 값들을 포함한다.

Default Value 42949672960

(125) <jms-server> <storage> <file-storage> <message-file> **<block-size>**

Description file storage 의 block 크기. 디폴트는 512Byte 이다.

Value Type nonNegativeIntType

Value Type Description 0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

Default Value 512

(126) <jms-server> <storage> <file-storage> <message-file> **<increase-rate>**

Description file storage 가 부족할 경우 storage 증가율을 지정. 디폴트는 50% 이다.

Value Type float

Default Value 5.0E-1

(127) <jms-server> **<jms-server-cluster>**

Description 데스티네이션 클러스터링을 사용하는 경우 여기에 참여하는 JMS 메시지 브로커에 대한 설정을 한다.

Child Elements
(128)cluster-service?
(129)jms-server-info+

(128) <jms-server> <jms-server-cluster> **<cluster-service>**

Description 클러스터링간 통신을 담당하는 서비스 이름을 지정한다. 지정하지 않을 경우 임의로 지정된다.

Value Type token

(129) <jms-server> <jms-server-cluster> **<jms-server-info>**

Description 클러스터에 참여하는 JMS 메시지 브로커에 대한 정보를 설정한다.

Child Elements
(130)broker-name
(131)broker-id
(132)ip-address
(133)jms-port?
(134)use-ssl?
(135)connection-interval?
(136)weight?

(130) <jms-server> <jms-server-cluster> <jms-server-info> **<broker-name>**

Description 클러스터간 통신에 사용되는 이름이다. 클러스터링에 참여하는 모든 JMS 메시지 브로커는 각각 유일한 값을 가져야 한다.

Value Type token

(131) <jms-server> <jms-server-cluster> <jms-server-info> **<broker-id>**

Description	클러스터간의 통신에 사용되는 브러커 아이디이다. 클러스터에 참여하는 모든 JMS 메시지 브로커는 각각 유일한 값을 가져야 한다. 값을 지정하지 않을 경우 cluster 테이블에 설정된 순서대로 임의 지정된다.
Value Type	positiveShortType
Value Type Description	JMS broker 의 borkerID 로 사용된다. 0 은 local brokerID 로 예약되어 있다.

(132) <jms-server> <jms-server-cluster> <jms-server-info> **<ip-address>**

Description	해당 JMS 메시지 브로커의 ip-address 주소값이다.
Value Type	token

(133) <jms-server> <jms-server-cluster> <jms-server-info> **<jms-port>**

Description	해당 JMS 메시지 브로커의 service port 값이다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.

(134) <jms-server> <jms-server-cluster> <jms-server-info> **<use-ssl>**

Description	SSL 을 사용할 것인지를 설정한다.
Value Type	boolean
Default Value	false

(135) <jms-server> <jms-server-cluster> <jms-server-info> **<connection-interval>**

Description	다른 클러스터 서버와의 connection 을 시도하는 주기이다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	60000

(136) <jms-server> <jms-server-cluster> <jms-server-info> **<weight>**

Description	server-weighted 방식의 메시지 분산 방식을 사용하는 경우 이 값이 사용된다.
Value Type	positiveIntType
Value Type Description	기본적으로 non-negative int 형이지만 1 이상의 값만 허용한다.
Default Value	1

B.4 Sample JMSMain.xml File

<<JMSMain.xml>>

```
<?xml version="1.0"?>
<jms-server xmlns="http://www.tmaxsoft.com/xml/ns/jeus”>
<jms-server>
   <service-config>
      <port>9741</port>
      <blocking-socket>false</blocking-socket>
      <connection-timeout>30</connection-timeout>
   </service-config>
   <thread-pool>
      <max>50</max>
   </thread-pool>
   <access-log>
      <level>INFO</level>
      <handler>
            <console-handler>
            <name>console</name>
            <level>INFO</level>
            </console-handler>
      </handler>
   </access-log>
   <durable-subscriber>
      <client-id>sports_client</client-id>
      <name>durable_sports</name>
      <destination-name>SportsTopic</destination-name>
      <message-selector>JMSType = 'sports' and age > 20</message-
selector>
   </durable-subscriber>
   <connection-factory>
      <type>queue</type>
      <name>ConFacSports</name>
      <export-name>ConFacSportsJndi</export-name>
      <client-id>client_sports</client-id>
```

```
   </connection-factory>
   <destination>
      <type>queue</type>
      <name>myDestName</name>
      <export-name>myDestJndi</export-name>
      <threshold-number-for-flow-control>3000</threshold-number-
for-flow-control>
      <multiple-receiver>true</multiple-receiver>
   </destination>
   <storage>
      <db-storage>
            <data-source-name>datasource1</data-source-name>
            <destination-table-name>jms_dest</destination-table-
name>
            <message-table-name>jms_message</message-table-name>
            <relation-table-name>jms_relation</relation-table-
name>
            <subscription-table-name>jms_sub</subscription-table-
name>
            <transaction-table-name>jms_tx</transaction-table-
name>
            <key-table-name>jms_key</key-table-name>
      </db-storage>
   </storage>

</jms-server>
```

색 인