JEUS JCA 안내서

Document info

Document name: JEUS JCA 안내서

Document date: 2005-06-06

Manual release version: 3

Software Version: JEUS 5

차 례

그림 목차

매뉴얼에 대해서

매뉴얼의 대상

본 문서는 JEUS 의 전반적인 기술에 대해서 모두 다루었다. 그러므로 JEUS 의 설치부터 설정, 모니터링 및 유지보수를 하는 관리자나 기타 관계자를 대상으로 한다.

매뉴얼의 전제 조건

본 문서를 읽기 전에 JEUS 소개나 JEUS 설치 안내서를 읽어보길 권한다. 그리고 JEUS 시작하기도 읽어보면 도움이 된다.

그리고, 어떤 장에서는 관련 내용의 기본 지식이 필요하다. 예를 들면, JNDI, JMX, Connector 등을 보기 위해서는 이들 기술에 대한 배경 지식이 있어야 한다. 이런 경우에는 다른 문서를 보거나 스펙을 읽어보길 바란다.

기본적으로 본 문서에서는 J2EE 와 Java 스펙에 대한 것은 설명하지 않고, JEUS 관련 내용만 설명한다.

매뉴얼의 구성

JEUS JCA 안내서는 다음처럼 6 장으로 구분되어 있다.

1. 소개

2. Connector 의 개요

3. Connector Packaging

4. Connection Management

5. Transaction Management

6. Security

일러두기

표기 예	내용
텍스트	본문, 12 포인트, 바탕체 Times New Roman
텍스트	본문 강조
CTRL+C	CTRL 과 동시에 C 를 누름
`public class myClass { }`	Java 코드
`<system-config>`	XML 문서
참조: / 주의:	참조 사항과 주의할 사항
JEUS_HOME	JEUS 가 실제로 설치된 디렉토리 예)c:\jeus50
jeusadmin nodename	콘솔 명령어와 문법
[파라미터]	옵션 파라미터
< xyz >	'<'와 '>' 사이의 내용이 실제 값으로 변경됨. 예)<node name>은 실제 hostname 으로 변경해서 사용
|	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.

표기 예	내용
. . .	XML 이나 코드 등의 생략
<<FileName.ext>>	코드의 파일명
그림 1.	그림 이름이나 표 이름

OS 에 대해서

본 문서에서는 모든 예제와 환경 설정을 Microsoft Windows™의 스타일을 따랐다. 유닉스같이 다른 환경에서 작업하는 사람은 몇 가지 사항만 고려하면 별무리 없이 사용할 수 있다. 대표적인 것이 디렉토리의 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
임시 커넥션	커넥션 풀이 꽉 찼을 때, 만들어져서 사용 후 폐기되는 커넥션
클러스터링	서로간에 연결된 컴포넌트의 그룹. 시스템을 더 안정적이고 효율적으로 만들 때 사용된다.
Base port	다른 포트 번호를 계산하는데 기본이 된다. 그리고 클라이언트가 JEUS Manager 로 접속하기 위해서도 설정한다. 기본값으로 9736

용어	정의
Container	런타임 어플리케이션 환경을 제공하는 실제 소프트웨어 구조
EIS	기존의 *Enterprise Information Systems*.
Engine	엔터프라이즈 어플리케이션에 서비스를 제공한다. J2EE 스펙에서 Engine 은 *Container* 라고 불린다.
Engine Container	JEUS 에서만 사용하는 개념으로, 여러 Engine 을 관리하는 단위이다. 각 Engine Container 는 자신의 JVM 에서 동작한다. 단 'default' Engine Container 는 JEUS Manager 와 동일한 JVM 에서 동작한다.
Group	JEUS 에서 사용되는 논리적인 개념으로, backup node 와 함께 사용된다. 대부분의 경우 노드와 동일하다.
J2EE	*Java 2 Enterprise Edition* 은 Sun Microsystems 에서 제정한 스펙들을 일컫는다. 이 스펙은 엔터프라이즈 어플리케이션을 Java 플랫폼으로 구현하는 것에 대해 정의하고 있다. JEUS 는 J2EE 1.4 스펙을 구현하고 있다.
JEUS	*Java Enterprise User Solution* 의 약자. JEUS version 5 는 J2EE 1.4 호환 WAS 이다.
JEUS Manager	JEUS Manager 는 JEUS 의 핵심 컴포넌트로 JEUS 노드를 관리하며, 다른 JEUS Manager 와 클러스터링을 구성한다.
JVM	*Java Virtual Machine* 의 약자

용어	정의
Load balancing	클러스터링에서 작업이 한쪽으로 몰리지 않도록 하는 기술
Node	JEUS 에서 사용되는 개념으로, 하나의 머신에서 작동하는 하나의 JEUS 를 뜻한다.
RA	Resource Adapter 의 약자
RM	Resource Manager 의 약자
Session	제한된 시간 동안 하나의 클라이언트에서 실행한 관련 작업 집합
TM	Transaction Manager 의 약자
TO (to)	TimeOut 의 약자
TX (Tx)	Transaction 의 약자
WAS	*Web Application Server* 의 약자. 복잡한 웹 어플리케이션을 실행하고 관리하는 미들웨어.
WebtoB	Tmax Soft 에서 만든 고성능 웹서버

연락처

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

USA
Tmax Soft, Co.Ltd.
2550 North First Street, Suite 110
San Jose, CA 95131
USA
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
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: 86-10-64106148 Fax: 86-10-64106144
E-mail : info@tmaxchina.com.cn
Web (Chinese): http://www.tmaxchina.com.cn

1 소개

J2EE Connector Architecture(이하 JCA)는 J2EE 플랫폼이 여러 이기종의 비 WAS 환경 서비스(이하 EIS, Enterprise Information System)에 연결할 수 있도록 하는 표준 아키텍처를 정의하고 있다.

JEUS WAS 는 인플로우 메세지 처리를 포함한 여러 가지 확장된 새로운 기능을 포함하는 JCA 1.5 스펙을 구현하고 있다.

이 JEUS JCA 안내서는 JEUS WAS 에서 구현된 스펙과 그 특징에 대해 설명한다. 또 Resource Adapter 를 개발하고 JEUS WAS 에서 Resource Adpater 를 패키징하고 디플로이 하는 방법에 대해 설명하고 있다.

2 Connector 의 개요

Java Connector Architecture (이하 JCA)는 J2EE 1.3 에서 JCA 1.0 으로 최초로 소개된 이후 J2EE 1.4 에 이르러 인플로우 메시지 처리를 포함한 여러 가지 확장된 기능을 포함하여 JCA 1.5 로 발전하게 되었다. 이번 장에서는 JEUS 5.0 에서 지원하는 JCA 컨테이너의 역할과 기능에 대해 설명한다.

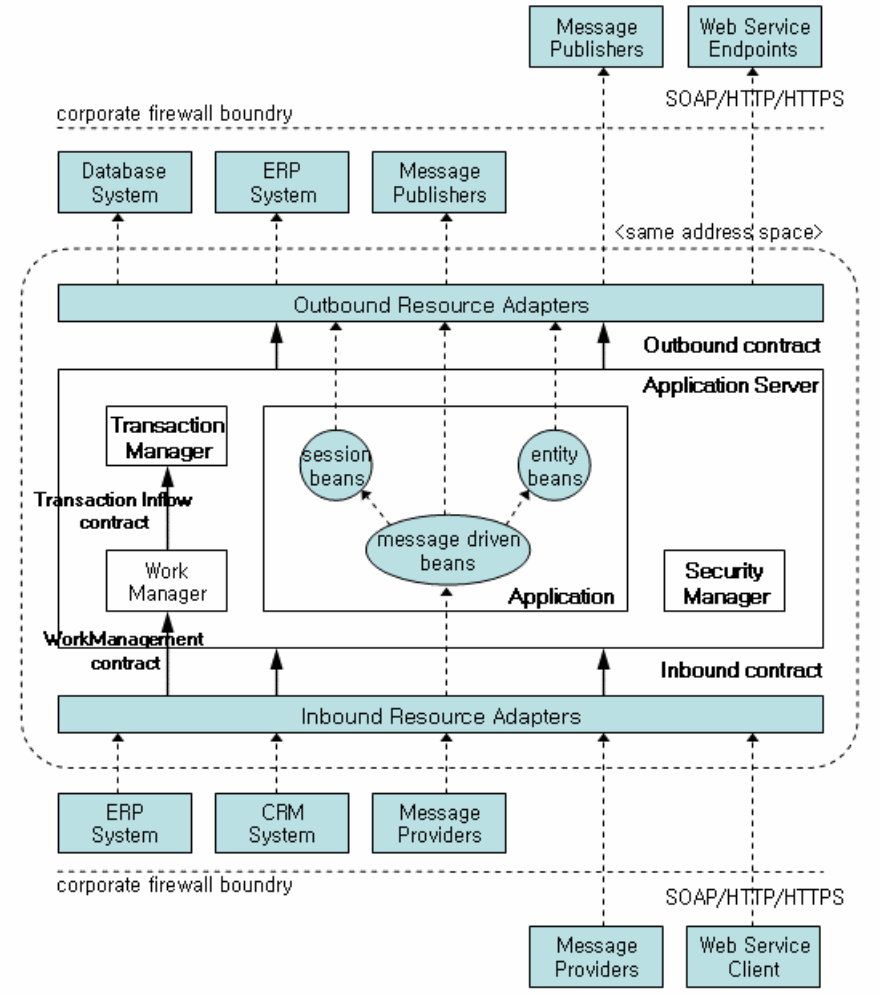

그림 1 JCA 아키텍처 개관

2.1 JCA Framework

JCA 는 ERP 나 Legacy 정보 시스템 등을 포함하는 비 WAS 환경 서비스(이하 Enterprise Information System=EIS)와의 연동을 위해 정의된 표준 프레임워크이다. JCA 이전까지는 이들 EIS 시스템과의 연동을 위해서 각 벤더의 EIS 및 WAS 마다 별도의 인터페이스를 구현하는 커스텀 드라이버를 구현하여 사용했으므로 특정 제품간에 연동 자체가 불가능 해지는 것을 제외하더라도 각각의 드라이버에 따라 다양한 인터페이스와 개발 방식을 따라야 하는 문제를 야기하였다.

이른바 N * M 문제라고 불려지는 이 상황은 기본적으로 코딩 레벨의 수정없는 연동을 불가능하게 만듦으로써 J2EE 환경의 이식성과 확장성에 심각한 제약을 가하게 되었다.

이문제를 해결하기 위하여 JCA 스펙은 리소스 어댑터와 WAS 컨테이너간의 인터페이스와 상호작용을 확정하여 표준에 맞는 WAS 와 리소스 어뎁터라면 코드 레벨의 수정없이 상호 호환성 있게 동작하는 것을 목표로 발전되었다. 각 EIS 벤더마다의 커스텀 드라이버를 표준화된 아키텍처로 대체함으로써 이전에 발생하였던 N * M 연동 문제를 N + M 으로 줄일수 있게 된다.

여기서 주목할 점은 Connector 아키텍처가 JDBC 아키텍처와 많은 면에서 비슷하다는 것이다. JDBC 는 관계형 DBMS 와의 연동 표준을 정의한다면, Connector 는 다양한 종류의 EIS(관계형 DBMS 를 포함하는)와의 연동 표준을 정의하고 있다. 어떻게 보면 Connector 는 JDBC 를 좀 더 광범위하고 일반화시킨 아키텍처라고 할 수 있다.

JEUS 에서 Connector 를 사용하기 전에 Connector 에 대한 기본적인 사항과 용어, 개념에 대해서 숙지할 것을 권한다. 개략적인 내용은 http://java.sun.com/j2ee/connector 에 소개되어 있고, Connector 1.5 스펙은 http://java.sun.com/j2ee/download.html#connectorspec 에서 받을 수 있다.

2.2 EIS integration

J2EE 1.4 와 EIS 시스템을 연동을 고려하는데 있어 이를 위한 도구로써 후보로 선택되는 것은 일반적으로 세가지를 든다. 앞서 설명한 JCA 와 자바 기반

의 메시징 서비스를 대표하는 JMS, J2EE 1.4 에 최초로 포함된 웹서비스가 그것이다. 이들은 사용 방법이나 서비스 형태의 상이함 이상으로 다양한 QOS 와 여러가지 장단점을 가진다.

- 웹서비스
  J2EE 에서 제공하는 서비스를 Services Oriented Architecture(이하 SOA)에 맞추어 구현한 것으로 서비스 제공자와 서비스 요구자 사이에 서비스 브로커가 존재한다. 서비스 제공자는 서비스 명세를 서비스 브로커에 등록하며 서비스 요구자는 이 브로커를 이용하여 원하는 서비스를 찾아 접근한다. 서비스 제공자와 서비스 요구자가 완전히 분리되어 있다는 측면에서 볼때 앞서 소개된 세가지 아키텍쳐들 중에서 가장 loosely-coupled 된 형태의 서비스를 제공한다. 자바 언어에 독립적인 플랫폼을 제공하며 HTTP 와 같은 널리 사용되는 프토콜을 이용하여 B2B 와 같은 넓은 영역의 서비스를 포괄할 수 있다. 현재 웹서비스 스펙은 트랜잭션을 지원하지 않는다.

- JMS
  자바 기반의 표준화된 메시징 인터페이스를 제공한다. 메시지 큐 / 토픽을 이용하여 포인트 투 포인트 방식(Point to Point)이나 퍼블리시/서브스크라이브(Publish/Subscribe) 방식을 지원하며 메시지를 보내는 측과 받는 측이 서로 분리되어 커플링이 적은 형태의 서비스 구성이 가능하다. 양방향 서비스가 가능하고 비동기 및 이벤트 방식의 메시지 처리, persistent 메시지를 지원하므로 비교적 시간이 오래 걸리는 비즈니스 플로우의 처리에 적합하며 JCA 와 웹서비스의 중간정도의 범위를 포괄할 수 있다. 자바 플랫폼에 한정되는 단점이 있으나 다른 메시지 기반 미들웨어와의 연동을 통해 일정부분 극복 가능하다

- JCA
  각 EIS 에 특정한 인터페이를 이용하는 가장 tightly-coupled 된 형태의 서비스를 제공한다. EIS 벤더에서 제공하는 리소스 어댑터를 통해 트랜잭션 전파, 시큐리티 인증과 같은 강력한 기능을 지원하지만 비동기형이나 이벤트 형태의 서비스 모델이 존재하지 않으므로 넓은 범위 혹은 시간이 걸리는 서비스에는 적합하지 않다. JCA 1.5 부터 양방향 서비스를 지원한다.

2.3 JCA Framework Composite

JCA 프레임워크를 구성하는 내용중에 EIS 리소스 어댑터 프로바이더와 WAS 제품의 컨테이너 프로바이더가 구현해야 하는 부분은 [그림 2]과 같다

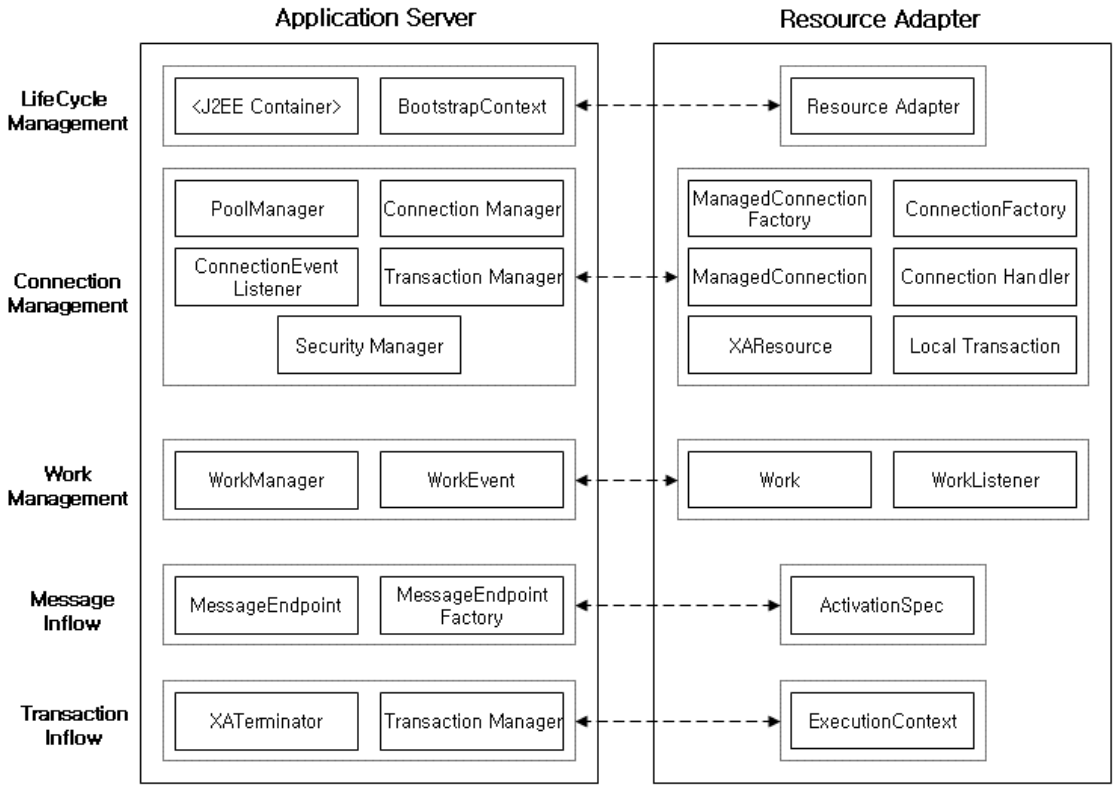

그림 2. JCA framework composites

JCA 1.5 는 외부 EAI 와의 연동을 위해 JCA 1.0 에서 이미 소개된 커넥션 매니지먼트 아키텍처(Connection Management) 외에 외부 EAI 에서 메시지를 받아들이기 위한 메시지 인플로우 아키텍처(Message Inflow)를 정의하며 이 두가지 기능을 포괄하기 위하여 이전에 개념적으로만 존재하였던 리소스 어댑터(Resource Adapter)를 하나의 인터페이스로 확정하였다.

리소스 어댑터는 리소스 어댑터 프로바이더에 의해 구현되어 독립적인 모듈로써 자체적인 라이프 싸이클을 가지며 EIS 에서 발생하는 메시지를 처리하기 위한 정보를 관리한다. 이러한 정보에는 액티베이션 스펙(Activation Spec) 과 메시지 엔드포인트 팩토리(Message Endpoint Factory)가 있다.

리소스 어댑터 프로바이더는 WAS 에서 기본적으로 제공하는 워크 매니저(Work Manager)와 타이머(Timer) 서비스를 사용하여 이러한 기능을 구현할 수 있다.

2.4 Lifecycle Management

리소스 어댑터의 라이프 싸이클은 리소스 어댑터 프로바이더가 구현한 리소스 어댑터를 통해 관리된다. 리소스 어댑터는 WAS 내에서 EIS 를 대표하며 WAS 와 EIS 간에 이루어지는 서비스의 준비 작업과 종료 작업을 위한 표준적인 인터페이스를 제공한다. WAS 는 이를 통해 리소스 어댑터의 시작 및 종료 명령을 내릴 수 있으며 리소스 어댑터는 이 시작/종료 명령을 받아 EIS 와의 통신 및 메시지 가공에 필요한 준비/종료 작업을 수행할 수 있다.

리소스 어댑터의 시작과 종료에 맞추어 WAS 내에 설정된 커넥션 풀이나 워크 매니저도 시작/종료된다.

[그림 3] 는 컨테이너와 리소스 어댑터간의 관계를 나타낸 것이다.

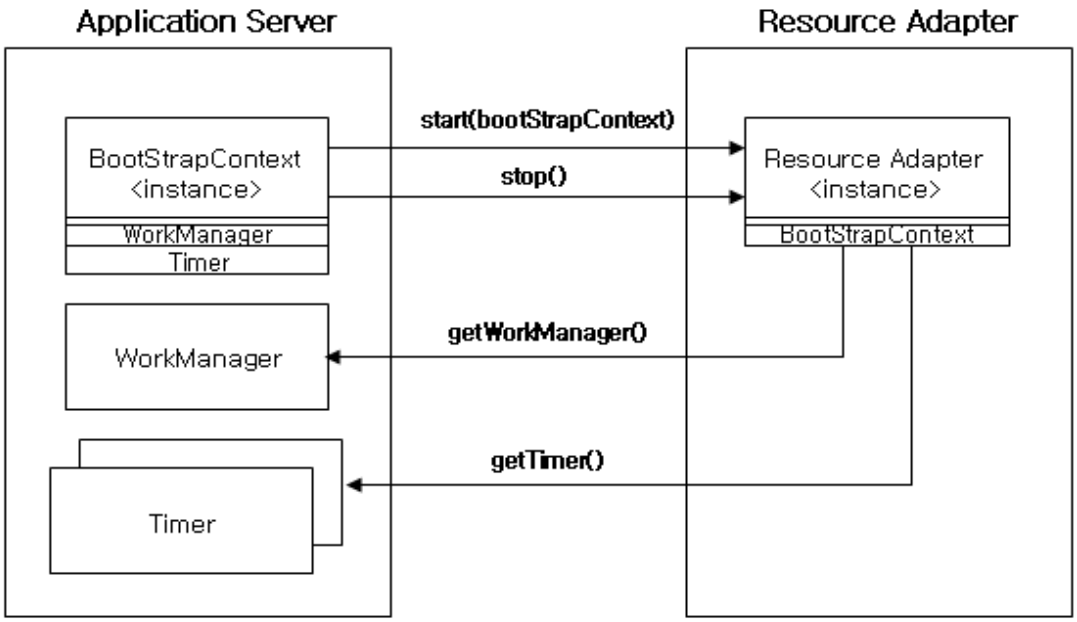

그림 3. Lifecycle Management

시작시에 컨테이너에서 리소스 어댑터로 전달되는 부트 스트랩 컨텍스트(BootStrapContext)는 리소스 어댑터에 워크 매니저와 타이머 서비스를 제공한다. 이들은 리소스 어댑터 내에서 쓰레드가 필요한 작업에 사용할 수 있다.

또한 리소스 어댑터는 EIS 에서 발생한 인플로우 메시지를 받아서 가공하는 역할을 수행하며 WAS 에서 넘기는 액티베이션 스펙과 메시지 엔드포인트 팩토리를 등록하는 대상이 된다.

부트스트랩 컨텍스트는 다음과 같이 정의된다.

```
public interface BootstrapContext {
      public Timer createTimer()
      public WorkManager getWorkManager()
      public XATerminator getXATerminator()
}
```

createTimer 메쏘드를 통해 생성되는 타이머는 리소스 어댑터에서 반복적인 작업을 원하는 경우에 사용할 수 있다. 리턴되는 타이머는 JDK 내의 타이머와 동일하므로 사용법은 JDK API 의 설명을 참조하도록 한다. 현재 JEUS 는 createTimer 가 호출 될때마다 타이머를 새로 생성하므로 리소스 어댑터 구현시에 너무 많은 타이머를 생성하지 않도록 한다.

getWorkManager 는 워크 매니저를 얻는 메쏘드이다. 자세한 내용은 1.7 절 Work Management 를 참조하도록 한다.

getXATerminator 는 EIS 에서 시작되어 전파된 트랜잭션을 처리하기 위한 트랜잭션 터미네이터(XATerminator)를 얻기 위해 사용된다. 자세한 내용은 Transaction Inflow Contract 를 참조하도록 한다.

JCA 1.0 기반의 리소스 어댑터의 경우처럼 리소스 어댑터가 아웃 바운드 서비스 만을 지원하는 경우 라이프 싸이클 매니지먼트 스펙은 반드시 구현되어야 할 필요는 없다.

2.5 Connection Management

커넥션 매니지먼트 아키텍처는 사용자의 요청에 따라 EIS 내의 작업을 위한 커넥션을 제공하고 관리하는 일반화된 표준을 정의한다. JCA 컨테이너는 기본적으로 커넥션 풀링을 지원하며 트랜잭션과 시큐리티 관련 작업을 유연하게 처리하기 위하여 WAS 의 시큐리티 매니저 및 트랜잭션 매니저와 유기적으로 연동되어 있다.

시큐리티 매니저와 트랜잭션 매니저는 리소스 어댑터와의 상호작용을 통하여 클라이언트가 원하는 작업을 수행하게 된다. JCA 는 WAS 내의 시큐리티 매니저 혹은 리소스 어댑터가 제공하는 커스텀 시큐리티 매니저를 사용할 수 있도록 규정하고 있다. 트랜잭션 매니저는 리소스 어댑터가 제공하는 트랜잭션 리소스를 관리하여 작업의 일관성을 유지하게 된다.

[그림 4] 은 클라이언트가 커넥션을 얻는 과정을 그림으로 나타낸 것이다.

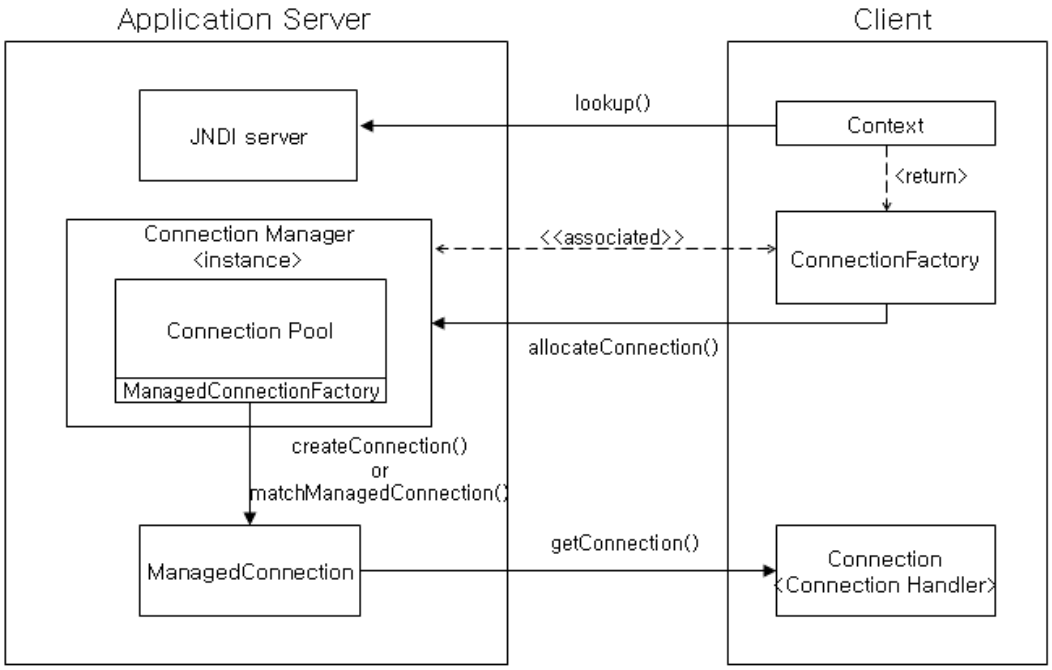

그림 4. Connection acquiring

클라이언트는 JNDI 에 등록되는 커넥션 팩토리(Connection Factory) 을 통하여 커넥션을 얻게 된다. 이 때 커넥션 팩토리는 내부적으로 연결되어 있는 JCA 컨테이너 내의 커넥션 매니저(Connection Manager) 에게 사용자의 요청을 전달하며 커넥션 매니저는 여기에 맞는 매니지드 커넥션을 찾아서 필요에 따라 시큐리티 인증과정이나 트랜잭션 관련 작업을 하게 된다. 커스텀 시큐리티 인증을 사용하는 경우 리소스 어댑터 프로바이더는 매니지드 커넥션(Managed Connection)에 필요한 인증작업을 구현하여야 한다.

매니지드 커넥션은 동시에 다수의 커넥션과 연관되어 있을수 있으며 새로 생성된 커넥션은 이전에 생성한 커넥션을 무효화 시키서는 않는다. 리소스 어댑터 프로바이더는 다수의 커넥션에 의해 하나의 매니지드 커넥션이 동시에 사용되는 경우에 대한 적절한 세만틱을 충족하여야 한다.

사용이 종료된 커넥션은 클라이언트에 의해 명시적으로 종료되어야 하며 리소스 어댑터는 이에 대해 적절한 커넥션 종료이벤트를 발생시켜야만 한다. JEUS 의 커넥션 매니저는 현재에 아무런 커넥션과도 연관되어 있지 않고 어떤 종류의 트랜잭션에도 참가하고 있지 않은 매니지드 커넥션에 대해서만 커넥션 풀링을 시도한다.

JCA 1.5 스펙은 이전 버전(1.0)에는 없었던 두가지 커넥션 최적화 방법을 제공하며 이는 트랜잭션과 밀접한 관계가 있으므로 다음의 Transaction Management 에서 설명한다.

2.6 Transaction Management

JEUS 트랜잭션 매니저는 JTA XAResource 기반의 분산 트랜잭션과 로컬 트랜잭션을 지원한다. 트랜잭션 내에 오직 하나의 2PC 리소스가 존재할 경우에는 1PC 커밋 최적화를 사용하며 하나의 분산 트랜잭션 내에 하나의 1PC 리소스를 허용하는 Last-Resource 커밋 최적화를 허용한다.

또한 로컬 트랜잭션에 관련되어 JCA 스펙에서 요구하는 커넥션 공유(Connection Sharing) 및 커넥션 연관(Connection Association) 기능을 제공한다. 커넥션 연관 기능을 사용하기 위해서는 커넥션 래퍼를 사용하여야 한다. 이 부분은 아래에 상세하게 설명한다.

2.6.1 트랜잭션 분리(Transaction Interleaving)

매니지드 커넥션은 JCA 컨테이너에 의해 관리되는 트랜잭션과 EIS 내부적으로 사용되는 트랜잭션 두가지로 구분할 수 있다. 리소스 어댑터 프로바이더는 EIS 내부적으로 사용되는 트랜잭션의 시작 및 종료 이벤트를 매니지드 커넥션에 등록된 이벤트 리스너를 통하여 컨테이너에게 알려주어야 하며 컨테이너는 이 이벤트를 받아 적절한 트랜잭션 분리(Transaction Interleaving)을 수행할 수 있게 된다.

EIS 로컬 트랜잭션이 이미 진행중인 매니지드 커넥션을 컨테이너 트랜잭션에서 다시 사용하려고 시도 하는 경우 컨테이너 트랜잭션은 시작되지 않으며 클라이언트는 Exception 을 얻게 된다. 반대로 컨테이너 트랜잭션이 진행중인 상황에서 EIS 로컬 트랜잭션이 시작되어 이벤트가 발생한 경우 컨테이너 트랜잭션은 롤백 마크되며 런타임 Exception 이 발생된다.

EIS 로컬 트랜잭션이 진행중인 매니지드 커넥션에 대해 유저 트랜잭션(UserTransaction)을 이용하여 컨테이너 트랜잭션을 시작려는 경우 Exception 이 발생하며 트랜잭션은 실패하게 된다. 세션빈 내에서 사용된 사용자 로컬 트랜잭션이 종료되지 않고 EJB 빈 풀로 반납되는 경우에도 트랜잭션은 롤백 처리되고 클라이언트는 Exception 을 받게 된다. 이때의 EJB 빈은 더이상 사용되지 않는다.

자동 커밋(Auto Commit)을 지원하는 커넥션이 글로벌 트랜잭션 내에서 사용되는 경우 리소스 어댑터 프로바이더는 트랜잭션 관리를 컨테이너가 담당할 수 있도록 자동 커밋 기능을 해제해 주어야 한다.

자동 커밋에 의해 시작/종료되는 EIS 트랜잭션 이벤트는 리소스 어댑터에 따라 이벤트가 발생하지 않을수도 있다. 컨테이너는 트랜잭션이 진행중이지 않은 것으로 생각하여 추가적인 컨테이너 트랜잭션을 적용할 수 있으며 이 경우의 트랜잭션 처리 결과는 전적으로 리소스 어댑터의 책임이다.

별도의 옵션을 사용하지 않는 경우 얻어진 커넥션은 커넥션 팩토리에서 커넥션을 얻는 순간의 트랜잭션 컨텍스트에 등록되게 된다. 커넥션을 얻은 이후에 발생된 트랜잭션 내에서 작업을 하기 원하는 경우 커넥션 래퍼(Connection Wrapper)를 사용하여야 한다. 이경우 클라이언트는 커넥션이 아닌 Deployment Descripter 내에 설정된 커넥션 인터페이스(Connection Interface) 를 구현한 래퍼를 얻게 되며 얻어진 래퍼에 대해 부적절한 캐스팅을 할 경우 Exception 이 발생 할 수 있으므로 주의하도록 한다.

늦은 트랜잭션 등록(Lazy Transaction Enlistment) 을 구현한 매니지드 커넥션에 대해서는 커넥션을 얻을 시점이 트랜잭션 범위내라도 트랜잭션 등록작업을 하지 않는다. 이경우 트랜잭션의 등록은 리소스 어댑터 프로바이더가 결정한다.

2.6.2 커넥션 공유(Connection Sharing)

두개 이상의 커넥션이 하나의 트랜잭션 내에서 하나의 EIS 리소스 매니저를 사용하는 경우 WAS 컨테이너는 효율적인 리소스 활용을 위해 커넥션 공유를 사용할 것인지를 재고한다. 컨테이너의 커넥션 매니저는 클라이언트가 커넥션을 얻을때 트랜잭션 컨텍스트를 조사하여 커넥션 공유가 가능한 상황인지를 판단한다.

커넥션 공유가 성립하기 위한 조건은 다음과 같다.

a. 동일한 트랜잭션
b. 동일한 VM 프로세스(동일한 주소공간)
c. 동일한 리소스 매니저를 사용
d. 동일한 커넥션 프로퍼티
e. 커넥션이 사용되는 어플리케이션의 공유 설정이 모두 shareable
f. 동일한 트랜잭션 관리 방법

글로벌 트랜잭션 외부에서 사용되는 커넥션에 대해서는 커넥션 공유를 사용하지 않으며, 트랜잭션을 지원하지 않는 커넥션의 경우도 마찬가지이다. 커넥션 공유는 리소스의 효율이라는 측면 외에도 리소스 매니저내에서 발생할 수 있는 락 컨텐션(Lock Contention) 이나 Read Isolation 문제를 피하는 데에도 효과적이다.

매니지드 커넥션에 대해 트랜잭션 격리 수준(Transaction Isolation Level) 이나 시큐리티 정보를 포함한 커넥션 프로퍼티의 수정 작업이 발생한 경우 리소스 어댑터는 사용자가 해당 매니지드 커넥션에 대해 커넥션을 얻으려는 시도를 할 때 커넥션 공유를 시도하지 못하도록 적절한 리소스 공유 Exception(SharingViolationException)을 발생시켜야 한다. 위와 같은 상황이 발생하였을때 리소스 어댑터가 리소스 공유 Exception 을 발생시켜야 하는 경우는 다음과 같다.

a. 해당 매니지드 커넥션이 두개 이상의 커넥션에 의해 사용중인 경우

b. 해당 매니지드 커넥션이 이미 로컬 혹은 분산 트랜잭션 내에서 사용중인 경우.

[그림 5] 는 커넥션 공유가 적용된 예이다.

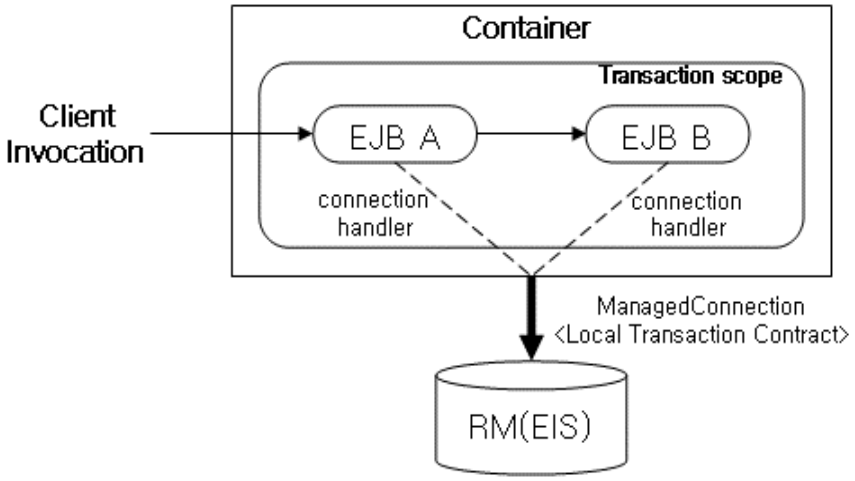

그림 5. Connection sharing

EJB A 와 EJB B 는 내에서 각각의 비즈니스 로직내에서 커넥션을 얻고 이를 이용하여 작업을 한다. EJB A 가 트랜잭션을 발생하고 EJB B 가 EJB A 의 트랜잭션 전파를 허용하는 경우(SUPPRTS) 두개의 EJB 는 하나의 트랜잭션 내에서 동일한 EIS 에 대해 작업을 하게 된다. 여기서 EJB B 에서 커넥션을 얻을때 컨테이너는 트랜잭션 컨텍스트를 조사하여 EJB A 에서 이미 얻은 매니

지드 커넥션이 있음을 확인하고 공유가 가능한 상황인지를 조사하여 동일한 매니지드 컨넥션에 대한 커넥션을 돌려주게 된다.

이 매니지드 커넥션은 두개의 커넥션으로 부터 작업을 수행하게 되며 트랜잭션이 종료되고 두개의 커넥션의 사용이 끝난후 (두개의 커넥션으로 부터 커넥션 종료 이벤트를 받은 후) 다시 컨테이너가 관리하는 커넥션 풀로 돌아가게 된다.

로컬 트랜잭션 밖에 지원하지 않는 매니지드 커넥션에 대해 커넥션 공유가 적용되지 않을 경우 두개의 작업은 별도의 매니지드 커넥션을 통해 작업이 이루어 지고 결과적으로 두개의 1PC 리소스가 하나의 트랜잭션 상에 존재하게 되므로 트랜잭션 매니저는 이를 에러 처리하게 된다. 두개의 커넥션의 상태가 앞서 설명된 대로 커넥션 공유가 가능한 상태라면 하나의 매니지드 커넥션과 하나의 1 PC 리소스만이 존재하게 되므로 트랜잭션은 성공하게 된다.

2.6.3 커넥션 연관(Connection Association)

커넥션을 얻는 과정에 적용되는 커넥션 공유와 달리 커넥션 연관은 이미 생성된 커넥션을 필요에 따라 매니지드 커넥션에 연관하는 작업을 의미한다. WAS 컨테이너는 어플리케이션의 라이플 싸이클 내에서 고정적으로 사용되는 커넥션에 대해 사용시점의 트랜잭션 범위에 맞추어 매니지드 커넥션을 바꾸게 되며 이러한 작업은 어플리케이션이나 사용자의 인지과정이 없는 상태에서 자동적으로 적용된다.

커넥션 연관이 적용된 예는 아래와 같다.

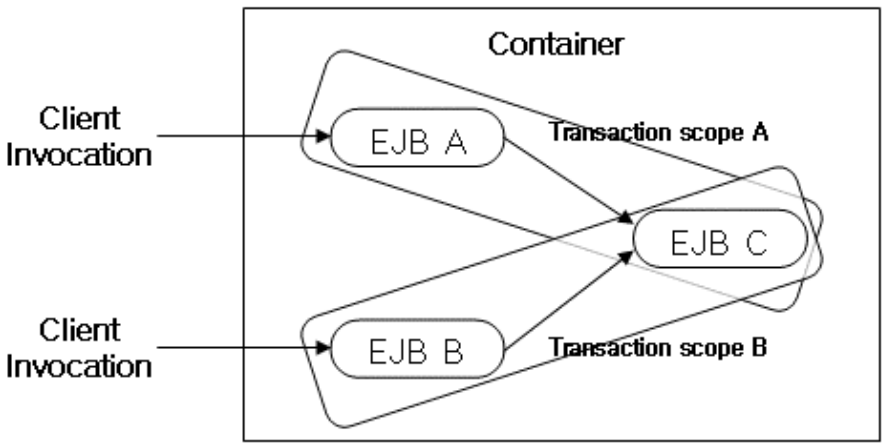

그림 6. Connection association

EJB C 는 처음 빈 생성과정에서 EIS 에 대한 커넥션을 얻은 후 빈의 라이프 싸이클 내에서 이를 종료하지 않고 계속 사용한다. EJB A 와 EJB B 는 트랜

잭션을 시작하여 같은 EIS 에 대해 커넥션을 얻어서 작업을 하고 EJB C 를 호출한다. EJB C 가 호출자의 트랜잭션 전파를 허용하는 경우(SUPPORTS) EJB A 와 EJB B 를 거쳐서 EJB C 에 적용되는 트랜잭션내에 사용중인 매니지드 커넥션은 호출시마다 다르게 된다. WAS 컨테이너는 이러한 상황에 맞추어 EJB C 의 커넥션을 EJB A 의 매니지드 커넥션 혹은 EJB B 의 매니지드 커넥션에 연관하는 작업을 한다. 이 등록 과정은 리소스 어댑터가 구현하는 매니지드 커넥션의 associateConnection 메쏘드를 통해 이루어진다.

```
interface javax.resource.spi.ManagedConnection {
      void associateConnection(Object connection) throws Resour
      ceException;
}
```

associateConnection 메쏘드가 호출되면 매니지드 커넥션은 인자로 넘겨진 커넥션이 현재 등록된 매니지드 커넥션과의 연결을 끊고 자신과 연관되도록 한다. 이때 이전의 매니지드 커넥션은 커넥션 매니저에 의해 커넥션 풀로 반납된다. 이 과정은 반드시 동일한 매니지드 커넥션 팩토리를 통해 생성된 매니지드 커넥션간에 일어나야 한다.

앞서 설명한 대로 커넥션 연관이 이루어지기 위해서는 커넥션 래퍼를 사용하여야 한다. 커넥션 래퍼는 커넥션의 메쏘드가 호출될 때마다 이러한 조건을 검사하게 되며 래퍼를 사용하지 않을때 보다 어느정도 효율이 떨어질 수 있다. 그러므로 사용자는 커넥션을 EJB 의 라이프 싸이클 내에서 계속 사용하지 말고 커넥션 팩토리를 사용하여 커넥션 연관보다 커넥션 공유를 사용하는 것을 권장한다.

2.6.4 로컬 트랜잭션 최적화(Local Transaction Optimize)

앞서 간단히 언급하였듯이 JEUS 는 JCA 스펙에서 설명하는 Last-Resource 커밋 최적화를 지원한다. 이는 하나의 글로벌 트랜잭션 내에 1PC 만을 지원하는 리소스를 하나 등록하는 것을 허용하여도 1PC 리소스의 작업을 포함하여 트랜잭션의 무결성을 담보할 수 있다는 것이다. Last-Resource 커밋 최적화는 TM 이 1PC 리소스를 제외한 다른 리소스의 트랜잭션 완료 준비작업(PREPARE)의 성공 이후에 해당 1PC 리소스에 대해 커밋을 수행하는 프로토콜을 의미한다.

일반 J2EE 를 운용하는 경우 하나의 트랜잭션 내에 많은 수의 리소스가 사용되지 않는다는 측면을 고려할 때 위의 프로토콜을 이용하면 2PC 를 사용하지 않고도 트랜잭션 작업을 완성할수 있거나 최소한 한번의 트랜잭션 준비

작업이 생략되어 약간의 성능상의 이득이 있을 수 있다. 즉 최초의 트랜잭션 시작시점에 2PC 리소스를 1PC 리소스로 등록하고 이후 트랜잭션 내에 추가적인 리소스가 등록된다면 그때 글로벌 트랜잭션을 시작하여 이후의 리소스는 모두 2PC 로 등록하는 것이다.

이것은 하나의 트랜잭션 내에 리소스의 종류가 적을 경우에 유용하며 다양한 종류의 리소스가 트랜잭션 내에 포함되는 경우는 큰 효과를 기대하기 힘들다. 이 최적화를 적용할때 주의할 점은 트랜잭션 내에 하나의 1PC 리소스가 등록된 상태에서 다른 1PC 만을 지원하는 리소스를 등록하려고 시도할 경우 TM 은 트랜잭션의 무결성을 확보할 수 없으므로 Exception 을 발생시키게 된다. 이러한 상황을 피하기 위해 JCA 리소스의 트랜잭션 지원 형태를 XATransactionOnly 로 설정할 수 있다. 이는 JCA 스펙이 지원하는 트랜잭션 형태가 아니고 JEUS 에 종속적인 것이므로 jeus-connector-dd.xml 에 설정하여야 한다. 자세한 내용은 Packaging 을 참조하도록 한다.

2.6.5 커넥션 최적화(Connection Optimization)

JCA 는 커넥션의 사용에 있어서 다음의 두가지 최적화 포인트를 제공한다. 이 기능들은 리소스 어댑터와 WAS 컨테이너간에 추가적인 이벤트를 교환하여 이루어지며 이를 위한 부가적인 인터페이스가 각각 매니지드 커넥션과 커넥션 매니저에 구현되어야 한다. JEUS 컨테이너는 이를 위한 인터페이스를 지원하며 리소스 어댑터가 생성하는 매니지드 커넥션이 이를 지원하는 경우 이를 자동적으로 사용한다.

- 늦은 커넥션 연관(Lazy Connection Association)

  사용자가 커넥션을 얻은후 장시간 사용하지 않는다면 이 커넥션은 실제 하는 일이 없이 점유된 형태로 남으며 이것은 리소스 관리측면에서 볼때 비효율적인 것으로 판단할 수 있다. 만약 리소스 어댑터에서 제공하는 매니지드 커넥션이 DissociatableManagedConnection 인터페이스를을 지원한다면 이렇게 장시간동안 사용되지 않은 커넥션을 풀로 돌리거나 다른 클라이언트에게 제공함으로써 좀더 효율적으로 리소스를 사용할 수 있게 된다.

  늦은 커넥션 연관에 관련된 인터페이스들은 다음과 같다.

```
**Connection Manager (implemented by JEUS container Provider)
interface LazyAssociatableConnectionManager {
```

```
      void associateConnection(Object connection,
ManagedConnectionFactory mcf, ConnectoinRequestInfo info)
throws ResourceException;
}

**ManagedConnectoin (implemented by Resource Adapter Provider)
interface DissociatableManagedConnection {
      void dissociateConnections() throws ResourceException;
}
```

컨테이너는 필요에 따라 DissociatableManagedConnection 을 호출할 수 있으며 매니지드 커넥션은 이 매니지드 커넥션에 연관되어 있는 모든 커넥션을 등록 해제 하여야 한다. 이 메쏘드가 호출된 이후에 커넥션이 재사용 될 경우 리소스 어댑터는 커넥션 매니저에 associateConnection 을 호출하여 새로이 얻어진 매니지드 커넥션에 이 커넥션을 등록하여야 한다.

- 늦은 트랜잭션 등록(Lazy Transaction Enlistment)

  어떤 로직에 트랜잭션이 진행중일 경우 이 트랜잭션 내에서 사용되는 모든 리소스들은 이 트랜잭션에 묶이게 된다. 그러나 만약 얻어진 커넥션으로 트랜잭션에 묶일 필요가 없는 작업만을 하였을 경우라면 이 커넥션은 실제 트랜잭션에 묶일 필요가 없다. 이렇듯 불필요하게 트랜잭션에 묶이는 경우 결과적으로 불필요한 리소스 관리 작업이나 트랜잭션 작업을 수행하게 된다.

  만약 리소스 어댑터가 트랜잭션에 묶이거나 묶일 필요가 없는 작업을 구분할 수가 있고 이를 커넥션 매니저에 알려준다면 불필요한 작업이 일어나지 않아도 되므로 성능이나 리소스 관리 측면에서 좀더 효율적으로 리소스를 사용할 수 있게 된다.

  늦은 트랜잭션 등록에 관련된 인터페이스들은 다음과 같다.

```
**Connection Manager (implemented by JEUS container Provider)
interface LazyEnlistableConnectionManager {
      void lazyEnlist(ManagedConnection) throws
ResourceException
}
```

```
**ManagedConnectoin (implemented by Resource Adapter
Provider)
interface LazyEnlistableManagedConnection {
}
```

LazyEnlistableManagedConnection 이 구현되어 있는 매니지드 커넥션에 대해 컨테이너는 트랜잭션에 관련된 검사 과정을 생략하며 리소스 어댑터가 명시적으로 lazyEnlist 를 호출하는 경우에만 현재의 트랜잭션에 매니지드 커넥션을 등록하는 작업을 한다.

2.7 Work Management

아웃바운드 메시지만을 처리하는 비교적 간단한 구조의 리소스 어댑터의 경우는 일반적으로 별도의 쓰레드를 필요로 하지 않으나 JCA 1.5 에 추가된 인바운드 메시지를 처리하기 위해 네트웍을 검사한다던지 하는 경우에 리소스 어댑터 내에서 쓰레드를 필요로 하는 경우가 있을 수 있다.

JCA 스펙은 리소스 어댑터 내에서 임의로 자바 쓰레드를 생성하지 말고 WAS 컨테이너가 제공하는 워크 매니저를 이용할 것을 할 것을 권장하고 있다. 이는 리소스 관리의 측면 뿐만 아니라 J2EE 시스템의 이식성을 높이는데 도움이 된다. JCA 스펙은 만약 리소스 어댑터가 반드시 자바 쓰레드를 써야만 하는 상황이라면 이를 데몬 쓰레드로 구현할 것을 권장하고 있다.

리소스 어댑터는 필요로 하는 작업을 워크 인스턴스로 구체화 하여 워크 매니저로 넘기게 되며 워크 매니저는 관리중인 쓰레드 풀에서 적당한 워커 쓰레드를 얻어 해당 작업을 수행 하게 한다. 리소스 어댑터는 워커 쓰레드가 수행하게 되는 작업의 시작/종료/Exception 등의 이벤트를 받기 원하는 경우 이벤트 리스너를 전달할 수 있고 또한 필요에 따라 해당 작업의 런타임 컨텍스트를 전달할 수도 있다.

런타임 컨텍스트는 트랜잭션 컨텍스트를 포함하며 이를 이용하여 EIS 에서 시작된 트랜잭션을 WAS 로 전파할 수도 있다. 자세한 내용은 Transaction Inflow Contract 를 참조하도록 한다.

2.7.1 워크(Work)

리소스 어댑터가 내용을 구현하여 워크 매니저에 전달하게 되는 워크 인터페이스는 JDK 의 Runnable 인터페이스를 확장한다. 사용자는 원하는 작업을 run 메쏘드 내에 구현하고, 작업 종료/리소스 반납을 위한 내용은 release 메쏘드 내에 구현하면 된다. 워크 매니저는 일정 시간이 지났을 때 혹은 사용자의 요청에 따라 이 메쏘드를 호출하게 된다. 리소스 어댑터는 나름대로의 방법으로 release 가 호출되었는지를 검사하여 추가적인 종료 작업을 수행할 수 있다. 구현된 run 와 release 메쏘드는 서로 동기화 메쏘드(synchronized method)로 선언되어서는 안된다.

```
public interface Work extends Runnable {
      public void release();
}
```

워크 매니저에 전달된 워크 인스턴스는 다음과 같은 상태를 거친다

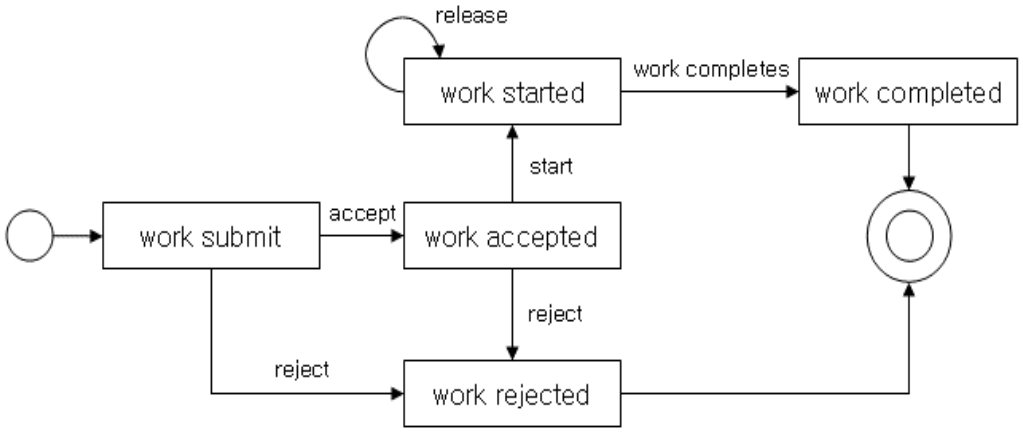

그림 7. Work State

작업이 워크 매니저에 전달된 시점에 ACCEPT 이벤트가 발생하며 워커 쓰레드에 의해 작업이 시작된 시점에 START 이벤트가 발생한다. 어떠한 이유로 지정된 시간 내에 작업이 시작될 수 없었다면 REJECT 이벤트가 발생한다.

작업이 시작된 이후는 Exception 등에 무관하게 모두 COMPLETE 이벤트가 발생한다.

2.7.2 워크 매니저

워크 매니저에는 리소스 어댑터가 WAS 컨테이너에서 관리하는 쓰레드 풀에 작업을 의뢰하기 위한 메쏘드가 정의되어 있다. 워크 매니저가 관리하는 쓰레드들은 모두 동일한 우선순위를 가진다.

워크 매니저는 아래와 같이 정의된다.

```
public interface WorkManager {
      public void doWork(Work work)
      public void doWork(Work work, long startTimeout, Executi
            onContext execContext, WorkListener workListener)
      public void scheduleWork(Work work)
      public void scheduleWork(Work work, long startTimeout, Exe
            cutionContext execContext, WorkListener workListener)
      public long startWork(Work work)
      public long startWork(Work work, long startTimeout, Execut
            ionContext execContext, WorkListener workListener)
}
```

각 메쏘드의 의미는 다음과 같다.

- doWork
  클라이언트는 작업이 완전히 종료될때까지 기다린다. 시작 시간을 지정한 경우 이 시간내에 작업이 시작(START)되지 않으면 작업 거부 Exception(WorkRejectedException)과 이벤트가 발생한다. 이때 Exception 코드값은 TIMED_OUT 으로 설정된다. 하나의 작업 내에서 doWork 를 이용하여 다시 작업을 시작한 경우(Nested Work) 내부에 시작된 작업이 종료된 이후에야 외부의 작업이 진행된다(LIFO). 지정된 시간내에 작업이 끝나지 않으면 release 메쏘드를 한번 호출해 준다.

- startWork
  작업이 시작될 때까지 기다린다. 시작 시간을 지정한 경우 이 시간내에 작업이 시작되지 않으면 작업 거부 Exception 과 이벤트가 발생하며 Exception 코드값은 TIMED_OUT 으로 설정된다.

- scheduleWork
  작업이 컨테이너에게 전달되면 바로 리턴한다. 시작 시간을 지정한 경우 이 시간내에 작업이 시작되지 않으면 작업 거부 Exception 과 이벤트가 발생하며 Exception 코드값은 TIMED_OUT 으로 설정된다. 이 메쏘드를

사용하는 경우 클라이언트는 대기 상태가 아닐수도 있으며 클라이언트는 이벤트만을 받게 된다.

리소스 어댑터 프로바이더는 반복적인 작업을 수행하기 위해서 워크 매니저와 타이머 서비스를 동시에 사용할 수 있다. 이러한 작업을 하는데에 있어 JCA 가 권장하는 코드 형식은 아래와 같다.

```
Timer timer = ra.bootstrap.createTimer();
WorkerManager manager = ra.bootstrap.getWorkManager();
timer.schedule( new TimerTask() {
   public void run() {
      try {
         manager.scheduleWork(new MyWork());
      } catch (WorkExceptoin we) {
         we.printStackTrace();
      }
   }, 0, 1000);
}
```

리소스 어댑터에서 전달된 런타임 컨텍스트(Execution Context)의 트랜잭션 정보가 부정확 할 경우 Transaction Inflow contract 에 의해 다음과 같은 Exception 이 발생할 수 있다. 자세한 내용은 Transaction Inflow contract 를 참조하도록 한다.

- 이미 같은 트랜잭션 아이디로 다른 쓰레드에 의해 진행중인 작업이 존재하는 경우. 이 때에는 WorkCompletedException 이 발생하며 이때 Exception 코드는 TX_CONCURRENT_WORK_DISALLOWED 가 된다.
- 해당 트랜잭션 아이디로 트랜잭션 컨텍스트를 구성하는데 실패한 경우. WAS 의 TM 이 어떤 특정 구조의 트랜잭션 아이디만 받아들이는 경우에 발생할 수 있다. 이 경우도 WorkCompletedException 이 발생하며 이때 Exception 코드는 TX_RECREATE_FAILED 가 된다.
- 이외에 트랜잭션에 관련되어 WAS 컨테이너 내부에서 발생한 모든 종류의 Exception 에 대해서 WorkCompletedException 이 발생하며 이때 Exception 코드는 INTERNAL 이 된다.

- 작업이 시작된 다음에 리소스 어댑터가 구현한 작업 내용 안에서 발생한 Exception 에 대해서도 WorkCompletedException 을 발생하며 이때 Exception 코드는 UNDEFINED 가 된다. 에러 메시지는 작업내에서 발생한 Exception 의 메시지가 된다.

2.7.3 워크 리스너(Work Listener)

리소스 어댑터가 이벤트를 받기 위해 전달하는 WorkListener 는 아래와 같이 정의된다. 이벤트는 각 상태에 해당하는 이벤트가 발생하면 워커 쓰레드에 의해 전달된다.

```
public interface WorkListener {
      public void workAccepted(WorkEvent e)
      public void workCompleted(WorkEvent e)
      public void workRejected(WorkEvent e)
      public void workStarted(WorkEvent e)
}
```

각 메쏘드는 워크 매니저에 의해 해당 이벤트가 발생되면 호출된다. 이벤트는 아래와 같이 정의된다.

```
public abstract class WorkEvent {
      public abstract WorkException getException()
      public abstract long getStartDuration()
      public abstract int getType()
      public abstract Work getWork()
}
```

getType 은 이벤트의 종류를, getWork 는 이벤트가 발생한 워크 인스턴스를 리턴한다. getException 은 작업 전/작업 중에 발생한 Exception 의 내용을 보여준다. 대표적으로 WorkRejectedException 와 WorkCompleteException 이 있다.

getStartDuration 은 작업이 ACCEPT 되어 START 되기 까지 걸린 시간을 리턴한다. 이 값은 정확한 시간을 보장하지 않으며 언제라도 워크 매니저는 UNKNOWN(-1)을 리턴할 수 있다.

2.8 Message Inflow Contract

JCA 1.5 는 JMS 를 포함하는 다양한 종류의 메시지 기반 서비스를 받아들일 수 있는 일반적인 메카니즘을 정의한다. 이 경우 메시징 서비스의 리소스 어댑터는 메시지 생산자, 엔드 포인트 어플리케이션은 메시지 소비자 역할을 맡게 된다. 전달되는 메시지의 타입은 WAS 컨테이너와 무관하며 실제적 메시지 전달이 이루어지는 메쏘드를 정의한 메시지 인터페이스는 리소스 어댑터가 임의로 결정할 수 있다.

[그림 8] 은 메시지 인플로우의 구성을 보여준다.

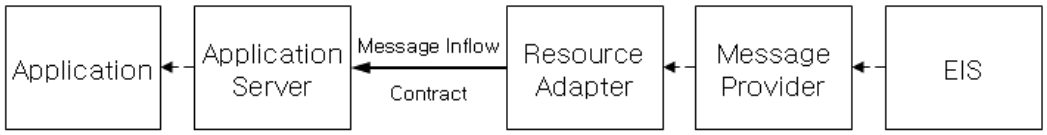

그림 8. Message Inbound Contract

메시지 생산자(message provider)는 메시지 소비자의 직접적인 요청에 의해서만 메시지를 전달하며 이 과정에 두 영역 사이에는 어떠한 컨텍스트의 전달 및 공유도 이루어 지지 않는다. 사용자가 EIS 에서 발생한 트랜잭션 컨텍스트를 J2EE 쪽에 전달하기 위해서는 워크 매니저를 이용하여 런타임 컨텍스트를 전달하여야 한다. 자세한 내용은 Transaction Inflow Contract 를 참고하도록 한다.

메시지 소비자(message consumer) 역할을 하는 엔드포인트 어플리케이션은 메시지 드리븐 빈 으로써 리소스 어댑터가 지원하는 메시지 인터페이스를 구현하여 개발되어야 한다. 엔드포인트 어플리케이션을 얻기 위한 메시지 엔드포인트 팩토리는 디플로이 과정에서 WAS 컨테이너에 의해 생성되어 리소스 어댑터에 제공된다.

엔드포인트 디플로이 관리자는 리소스 어댑터의 Deployment Descriptor 에 설정된 메시지 인터페이스나 액티베이션 변수의 값, QOS 등을 기반으로 원하는 리소스 어댑터를 결정하고 해당 리소스 어댑터에 맞추어 메시지 드리븐 빈의 Deployment Descriptor 에 적절한 액티베이션 스펙값을 지정하여야 한다. 엔드포인트 어플리케이션은 필요에 따라서 디플로이 과정에 Administered Object 를 네이밍 서버에 등록할 수도 있다.

[그림 9] 은 메시지가 전달되는 과정을 나타낸 것이다.

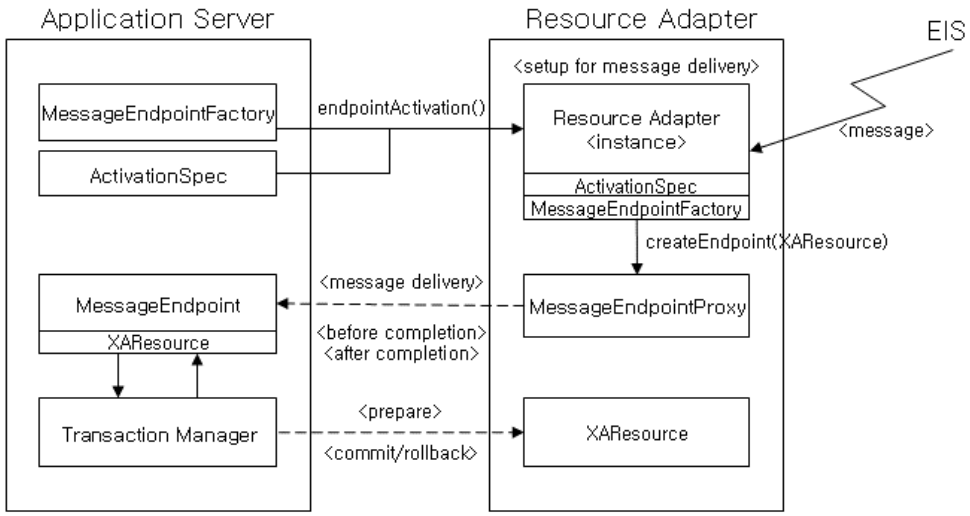

그림 9. Message Inbound Contract

엔드포인트 어플리케이션은 Deployment Descriptor 에 설정된 액티베이션 변수의 값을 이용하여 메시지를 받고자 하는 리소스 어댑터의 액티베이션 스펙을 생성하며, 이 액티베이션 스펙과 메시지 엔드포인트 팩토리를 리소스 어댑터 측에 넘긴다. 이 과정은 해당 엔드포인트 어플리케이션의 디플로이 과정에서 컨테이너에 의해 이루어진다.

리소스 어댑터는 넘겨진 액티베이션 스펙을 기반으로 메시지 생성 및 전달을 위한 힌트를 얻게 된다. 리소스 어댑터에 메시지가 도착하면 디플로이 과정에 얻어진 엔드포인트 팩토리를 이용하여 메시지 엔드포인트를 생성하며 여기에 대해 메시지 인터페이스를 이용하여 메시지를 직접 전달한다.

엔드포인트 어플리케이션은 메시지 인터페이스에 적절한 트랜잭션 정책을 설정할 수 있으며 리소스 어댑터는 실제 전달된 메시지가 사용되는 과정에 사용되는 트랜잭션에 관련된 이벤트를 얻기 위해 메시지 전달 과정에 XAResource 를 등록할 수도 있다. 등록된 XAResource 는 WAS 트랜잭션 매니저에서 트랜잭션 작업을 수행할때 자동적으로 호출되며 리소스 어댑터는 트랜잭션 이벤트를 받아서 원하는 작업을 수행할 수 있다. 리소스 어댑터에서 XAResource 를 등록 하더라도 메지시 전달에 사용된 메시지 인터페이스 메쏘드가 트랜잭션을 지원하지 않는 경우(NOT-SUPPORTED) 트랜잭션 이벤트는 발생하지 않는다. 또한 Transaction inflow contract 를 통해 EIS 에서 전파된 트랜잭션이 메시지 전달에 사용되는 경우에도 등록된 XAResource 는 무시된다. XAResource 에서 발생한 이벤트를 이용하여 작업을 할 경우

J2EE 내에서 해당 XAResource 가 사용되는 시기나 방법에 대해서 어떠한 가정도 있어서는 안된다.

실제 메시지 전달의 트랜잭션 처리에 있어서 리소스 어댑터는 두가지 방법을 취할 수 있다. 하나는 컨테이너에게 트랜잭션 시작과 종료를 일임하는 것이고, 하나는 메시지 엔드포인트의 beforeDelivery, afterDelivery 를 메시지 전달 전후로 호출하여 리소스 어댑터에서 트랜잭션 범위를 명시적으로 지정하는 것이다. 두번째 방법을 쓰는 경우 리소스 어댑터는 반드시 XAResource 를 등록하여야 하며 그렇지 않은 상태에서 호출된 beforeDelivery 와 afterDelivery 는 아무런 의미를 가지지 않는다. 리소스 어댑터는 같은 쓰레드 내에서 beforeDelivery 와 afterDelivery 를 호출하여야 하며 이를 위반한 경우 InvalidStateException 이 발생하고 트랜잭션은 롤백 처리된다. beforeDelivery, afterDelivery 중간에 release 가 호출된 경우 beforeDelivery 에서 새롭게 시작된 트랜잭션은 롤백 처리된다.

엔드포인트 어플리케이션은 하나의 메시지 생산자에 대해 등록되며 리소스 어댑터는 다수의 동일/유사한 액티베이션 스펙에 대해 각각 독립적인 메시지 전달을 보장하여야 한다. 또한 하나의 메시지 엔드포인트에 대해 동시에 두개 이상의 메시지를 전달해서는 안된다. 다만 별도의 메시지 엔드포인트에 동시에 메시지를 전달하는 것은 허용된다.

엔드포인트 어플리케이션으로 개발되는 메시지 드리븐 빈은 다른 EJB 를 호출하기 위한 디스패처(Dispatcher)로 활용할 수 있다. 이경우 리소스 어댑터가 전달하는 메시지는 EJB 의 호출에 필요한 정보가 되며 메시지 드리븐 빈과 리소스 어댑터는 밀접한 관계를 맺게 된다. 이 메시지 드리븐 빈은 리소스 어댑터 프로바이더에 의해 구현되어 리소스 어댑터의 한 부분으로써 같이 패키징/디플로이 될수 있으며 이때 이 메시지 드리븐 빈은 로컬 오브젝트로 개발되는 것이 유리하다.

2.9 Transaction Inflow Contract

외부 트랜잭션 매니저에서 시작된 트랜잭션 내에서 작업을 하기를 원하는 경우 리소스 어댑터는 WAS 컨테이너에게 트랜잭션을 전파할수 있어야 한다. 이를 위해 JCA 1.5 는 워크 매니저를 이용한 트랜잭션 컨텍스트 전파를 가능하도록 하고 있다. 또한 장애 상황에서 트랜잭션 정합성을 보장하기 위하여 EIS 측의 리커버리 프로토콜을 정의하고 있다.

리소스 어댑터가 컨텍스트를 전달하기 위한 런타임 컨텍스트는 아래와 같이 정의된다. 전달된 트랜잭션 타임아웃 시간과 트랜잭션 아이디는 WAS 컨테이너에 의해 사용된다. 앞장에 잠깐 언급되었듯이 하나의 트랜잭션 아이디는 동시에 최대 하나의 워커 쓰레드에 의해서만 작업을 진행 할 수 있다. 이를 지키지 않을 경우 WorkCompletedException 이 발생하며 Exception 코드는 TX_CONCURRENT_WORK_DISALLOWED 가 된다.

```
public interface ExecutionContext {
      public void setTransactionTimeout(long timeout)
      public long getTransactionTimeout()
      public void setXid(Xid xid)
      public Xid getXid()
}
```

리소스 어댑터는 EIS 에서 전달되는 트랜잭션 컨텍스트와 프로토콜을 정확히 분석하여 워크매니저에 전달하여야 한다. 컨테이너가 리소스 어댑터에서 넘긴 컨텍스트를 기반으로 트랜잭션을 재구성하지 못할 경우 위와 동일한 Exception 이 발생하며 이경우 Exception 코드는 TX_RECREATE_FAILED 가 된다.

리소스 어댑터가 EIS 에서 전파된 트랜잭션을 처리하기 위한 트랜잭션 터미네이터(XATerminator)는 앞서 설명한 대로 리소스 어댑터의 시작 시간에 부트스트랩 컨텍스트를 통해 리소스 어댑터로 전달된다. 트랜잭션 터미네이터는 트랜잭션을 완료하기 위한 메쏘드가 정의되어 있으며 아래와 같이 정의된다.

```
public interface XATerminator {
      public void commit(Xid xid, boolean onePhase)
      public void forget(Xid xid)
      public int prepare(Xid xid)
      public Xid[] recover(int flag)
      public void rollback(Xid xid)
}
```

EIS 에서 전파된 트랜잭션이 미처 종료되지 않은 상태에서 EIS 혹은 WAS 에서 장애가 발생한 경우 트랜잭션의 정합성을 보장하기 위하여 EIS 와 WAS 는 다음과 같은 리커버리 프로토콜을 따른다.

- 트랜잭션이 활성화된한 상태인 경우, 즉 트랜잭션이 시작되고 준비 메시지(PREPARE)가 전달되기 이전에 EIS 혹은 WAS 에서 장애가 발생한 경우 : EIS 및 WAS 는 해당 트랜잭션이 실패한 것으로 간주하여 즉시 트랜잭션을 롤백 처리하고 리턴한다(Unilateral / Presumed Rollback Protocol). JCA 는 이 작업이 리소스 어댑터 내에서 이루어 지는 것을 권장하고 있다.

- 트랜잭션 준비과정을 마치고 완료되기 이전 상태인 트랜잭션 인다우트(in-doubt) 상태에서 WAS 가 다운된 경우 : EIS 는 WAS 쪽에 계속 재접속을 시도하며 이것이 성공하면 트랜잭션 결과에 맞게 커밋/롤백을 시도하면 된다. EIS 와 리소스 어댑터 간의 리커버리 프로토콜은 리소스 어댑터 프로바이더가 임의로 구현한다.

- 트랜잭션 인다우트 상태에서 EIS 에서 장애가 발생한 경우 : WAS 는 EIS 쪽으로 재접속을 시도하며 이것이 성공하면 리소스 어댑터는 트랜잭션 터미네이터에 대해 recover 메쏘드를 호출하며 여기에 대해 WAS 는 현재 인다우트상태에 있는 트랜잭션 아이디의 리스트를 리턴한다. EIS 는 이를 받아 트랜잭션 결과에 맞게 커밋/롤백을 시도한다. EIS 와 리소스 어댑터 간의 리커버리 프로토콜은 리소스 어댑터 프로바이더가 임의로 구현한다.

2.10 Common Client Interface

Common Client Interface(이하 CCI) 는 모든 리소스 어댑터의 아웃바운드 서비스를 사용하는 데 있어 통일된 API 를 제공하자는 취지로 발전된 것이다. CCI 를 이용함으로써 개발 방식을 일관되게 유지할 수 있으며 무엇보다 툴을 이용한 손쉬운 개발이 가능해진다. CCI 는 현재 널리 쓰이고 있는 JDBC 와 유사하며 이의 확장형태로 볼 수 있다. CCI 는 다양한 종류의 서비스 형태를 제공하며 EIS 에 종속적인 인터페이스를 사용하지 않고도 EIS 종속적인 데이터를 다룰수 있는 장점이 있다.

JCA 1.5 는 리소스 어댑터 개발시 CCI 를 사용할 것을 권장하고 있다. CCI 는 서비스의 호출을 인터액션(Interaction) 으로 개념화 하여 서비스 방식에 관한 내용은 인터액션 스펙(Interaction Spec) 이라 불리는 EIS 에 종속적인 인스턴스로 추상화한다. 서비스 호출의 인자와 리턴값은 레코드 형식으로 제공된다.

인터액션 스펙은 일반적으로 서비스 함수 이름(Function Name)과 서비스 형식(InteractionVerb) 두가지를 포함할 것을 권장한다. CCI 는 동기화 보내기, 동기화 받기, 동기화 보내고 받기 세가지 서비스 형식을 지원하며 비동기 서비스는 지원하지 않는다.

아래는 CCI 인터페이스를 이용하여 EIS 의 서비스를 호출하기 위한 준비과정이다.

```
**get Connection to EIS by lookup ConnectionFacotry in JNDI
javax.naming.Context nc = new InitialContext();
javax.resource.cci.ConnectionFactory cf = (ConnectionFactory)nc.
               lookup(“java:/comp/env/eis/ConnectionFactory”);
javax.resource.cci.Connection cx = cf.getConnection();

**create Interaction
javax.resource.cci.Interaction ix = cx.createInteraction();

**create Interaction Spec
com.wombat.cci.InteractionSpecImpl spec = .....
spec.setFunctionName(“<EIS_FUNCTION_NAME>”);
spec.setInteractionVerb(InteractionSpec.SYNC_SEND_RECEIVE)”
....
```

서비스에 필요한 데이터를 레코드에 저장하여 다음과 같이 서비스를 호출한다.

```
**call EIS service
boolean ret = ix.execute(spec, input, output);
```

3 Connector Packaging

이번장에서는 개발된 리소스 어댑터를 패키징하고 Deploy 하는 과정을 설명한다. JCA 는 J2EE 호환 어플리케이션 서버에 이식성이 있고 모듈화된 방식의 Deploy 를 지원한다.

3.1 Overview

리소스 어댑터는 JCA 스펙에 맞게 구현된 여러가지 자바 인터페이스 및 클래스 파일과 리소스 어댑터에 필요한 기본적인 Deployment Descriptor 파일을 포함한다. 필요에 따라 네이티브 라이브러리나 문서파일, 그림파일 등이 여기에 추가되기도 한다. 이 파일들은 J2EE 내에서 리소스 어댑터 모듈이라는 단위를 구성하게 되며 JMX 관리하에서 작동하게 된다.

리소스 어댑터 모듈은 두가지 형태로 Deploy 될 수 있다.

- 어플리케이션 내에 독립적인 모듈로써 Deploy 되는 경우. 이경우 리소스 어댑터 모듈은 해당 J2EE 서버 모듈하의 모든 어플리케이션에서 공동으로 사용할 수 있게 된다.

- 한가지 이상의 다른 J2EE 모듈로 구성되는 J2EE 어플리케이션의 한 구성원으로써 Deploy 되는 경우로 J2EE 어플리케이션은 별도의 어셈블리 파일을 이용하여 구성된다. 이 경우 해당 리소스 어댑터 모듈은 J2EE 어플리케이션 내에서만 사용할 수 있다.

3.2 Packaging

리소스 어댑터 모듈은 Java Archive 포맷(JAR)을 사용하여 RAR 를 확장자로 사용한다. RAR 파일에는 JCA 가 지정한 포맷대로 설정된 Deployment Descriptor 파일이 존재하여야 하며 이 파일은 META-INF/ra.xml 위치에 있어야 한다. RAR 파일 내에는 클래스 파일, 이미지 파일, 문서 파일과 JAR 파일이 존재 할 수 있다. 이들은 RAR 파일 내 임의의 위치에 존재할 수 있다.

ra.xml 에 설정되는 내용은 스키마 파일 connector_1_5.xsd 을 참조하도록 한다. 이 파일에 설정되는 내용은 대략 다음과 같다.

- 리소스 어댑터 자체에 관한 내용. (리소스 어댑터의 이름과 버전, 개발사, UI 아이콘의 URL, 라이센스 정보, EIS 시스템에 대한 정보들)

- 리소스 어댑터 클래스에 관련된 내용. (구현된 리소스 어댑터의 클래스 이름, 리소스 어댑터의 프로퍼티들)
- 아웃 바운드 메시지 처리에 관련된 내용. (매니지드 커넥션 팩토리, 커넥션 팩토리, 매니지드 커넥션, 커넥션, 커넥션 인터페이스의 클래스 이름, 매니지드 커넥션 팩토리의 프로퍼티들, 지원하는 트랜잭션 종류, 시큐리티 인증 방식)
- 인바운드 메시지 처리에 관한 내용. (메시지 리스너 인터페이스, 액티베이션 스펙의 클래스 이름, 필수적인 액티베이션 스펙의 내용)
- Administered Object 에 관련된 내용.

ra.xml 이외에 JEUS 에 종속적인 환경을 구성하기 위하여 별도의 Deployment Descriptor 를 추가로 생성한다. 이 파일은 META-INF/jeus-connector-dd.xml 에 위치한다. 이 파일에는 다음과 같은 내용이 설정된다.

- 워크 매니저를 위한 쓰레드 풀의 설정
- 아웃바운드 커넥션 풀을 위한 설정.

3.2.1 jeus-connector-dd

- module-name : 해당 리소스 어댑터의 J2EE 리소스 어댑터 모듈 이름.
- worker-pool : 워크 매니저를 사용할 경우 쓰레드 풀에 대한 설정 값.
- connection-pool : 아웃 바운드 서비스를 사용할 경우 커넥션의 프로퍼티와 커넥션 풀에 관련된 설정 값.

3.2.2 worker-pool

- min [default : 3] : 워크 매니저가 유지하는 최소 워커 쓰레드의 개수.
- max [default : 10] : 워크 매니저가 제공하는 워커 쓰레드의 최대 개수.
- step [default : 1] : 워커 쓰레드를 새로 생성할 필요가 있을 경우 증가하는 크기.
- pre-allocation [default : true] : true 로 하면 리소스 어댑터가 시작될 때 워크 매니저는 앞에 설정된 min 만큼의 워커 쓰레드를 미리 생성한다. false 일 경우는 사용자의 요청이 있을 때 생성한다.

- submit-timeout [default : 5000] : 리소스 어댑터에서 워크 매니저로 작업을 의뢰했을 때 이 시간이 지나도록 가용한 워커 쓰레드가 없을 경우 해당 작업은 REJECT 되며 Exception 이 발생한다. msec 단위로 설정하며 0 으로 설정하면 계속 기다리게 된다.

- release-timeout [default : 60000] : doWork 메쏘드에 의해 워커 쓰레드에 의해 시작된 작업이 이 설정 값의 시간이 지나도 종료되지 않을 경우 워크 매니저는 해당 작업에 대해 release 를 한번 호출하게 된다. msec 단위로 설정하며 0 이면 이를 사용하지 않는다.

- shutdown-timeout [default : -1] : 리소스 매니저를 종료할 때 현재 워커 쓰레드에 의해 진행중인 작업이 있다면 워크 매니저는 최대 이 시간만큼 기다렸다가 종료한다. msec 단위로 설정하며 -1 일 경우 계속 기다린다.

- pooled-timeout [default : 600000] : 워크 매니저에서 관리중인 워커 쓰레드의 수가 앞서 설정된 최소값보다 크고 풀에서 대기중인 워커 쓰레드가 이 시간 동안 사용되지 않았다면 해당 쓰레드를 풀에서 제거한다. msec 단위로 설정하며 0 이면 무시된다.

3.2.3 connection-pool

- export-name [mandatory] : 커넥션 팩토리를 찾기 위한 JNDI 네임이다.

- transaction-support [optional] : 해당 풀의 커넥션이 지원하는 트랜잭션 타입을 지정한다. 여기에 지정된 값은 ra.xml 에 설정된 값보다 우선시 된다. 표준 세가지 타입에 XATransactionOnly 를 추가하여 다음 네 가지를 지정할 수 있다.

  - NoTransaction : 트랜잭션을 지원하지 않는 커넥션이다.

  - LocalTransaction : 로컬 트랜잭션만을 지원하는 커넥션이다.

  - XATransaction : XA 트랜잭션과 로컬 트랜잭션을 모두 지원한다.

  - XATransactionOnly : XA 트랜잭션만을 지원한다. JCA 표준은 아니며 몇몇 특수한 EIS 를 사용하는 경우와 한 트랜잭션 내에서 다른 로컬 트랜잭션 리소스가 같이 사용될 경우 설정한다. 이 값을 설정하면 해당 커넥션에 대해 로컬 트랜잭션 최적화를 시도하지 않는다. 자세한 설명은 Local Transaction Optimize 를 참조하도록 한다.

- user [optional] : 시큐리티 인증 타입을 컨테이너로 설정한 경우에 사용된다. 여기에 설정된 값은 ra.xml 에 설정된 값보다 우선시 된다.

- password [optional] : 위와 마찬가지로 시큐리티 인증 타입을 컨테이너로 설정한 경우에 사용된다. 여기에 설정된 값은 ra.xml 에 설정된 값보다 우선시 된다.

- use-wrapper [default :true] : 리소스 어댑터가 구현한 커넥션에 대한 래퍼를 사용한다. 이를 이용하면 추가적인 상태 추적이 가능해지며 커넥션 연관을 사용 할 수 있게 된다.

- dissociation-timeout [default : 0] : 커넥션 풀이 제공하는 매니지드 커넥션이 DissociatableManagedConnection 을 지원하는 경우에만 의미있는 값이다. 지정된 시간을 초과하여 사용되지 않은 커넥션에 대하여 연관관계를 해제하는 작업을 한다. 이 작업의 판단 기준은 래퍼를 사용할때와 사용하지 않을 때의 의미가 조금 다른데 래퍼를 사용하는 경우에는 매니지드 커넥션에 마지막으로 접근한 시간을 기준으로 하며 래퍼를 사용하지 않은 경우에는 최초에 커넥션을 얻은 시간을 기준으로 한다. msec 단위로 설정하며 0 이면 무시된다.

- invalidation-timeout [default : 0] : 클라이언트에 의해 사용중인 매니지드 커넥션이 지정된 시간을 초과하여 커넥션 풀에 반납되지 않을 경우 해당 매니지드 커넥션에 대해 destory 를 시도한다. 위와 마찬가지로 래퍼를 사용하지 않을 경우는 최초에 커넥션을 얻은 시간부터의 경과시간이 되고, 래퍼를 사용하는 경우는 매니지드 커넥션에 마지막으로 접근한 시간으로부터의 경과시간을 기준으로 한다. msec 단위로 설정하며 0 이면 무시된다.

- skp-connection-matching [default :false] : 기본적으로 JCA 스펙은 매니지드 커넥션을 사용하기 전에 반드시 matchManagedConnection 메쏘드를 이용하여 리소스 어댑터의 인증을 시도하여 성공했을 경우에만 사용할 수 있도록 규정하고 있다. 이 과정을 생략하고자 하는 경우에는 이 값을 true 로 설정한다.

- pool-management : 매니지드 커넥션 풀에 관련된 설정이다.

3.2.4 pool-management

- min [default : 0] : 커넥션 풀은 이 값보다 작은 매니지드 커넥션은 계속 유지하게 된다.

- max [default : 5] : 커넥션 풀이 관리할수 있는 최대 매니지드 커넥션의 수이다.

- pooled-timeout [default : 600000] : 커넥션 풀에 의해 관리되는 매니지드 커넥션의 수가 위에 지정된 min 보다 클 경우 풀에서 대기중인 커넥션이 이 시간 동안 사용되지 않았다면 풀에서 제거한다. msec 단위로 설정하며 이 값을 0 으로 설정하면 커넥션을 계속 유지하게 된다.

- wait-connection : 커넥션 풀에 가용한 매니지드 커넥션이 없을 경우에 대한 설정값이다.

- disposable-connection : 커넥션 풀에 가용한 매니지드 커넥션이 없고 위에 설정된 반납 대기시간도 초과했을 경우 해당 클라이언트가 임시로 사용하기 위한 커넥션을 생성할 것인지를 설정한다. 이렇게 새성된 커넥션 커넥션 풀의 관리를 받지 않으며 클라이언트에 의해 한번만 사용되고 제거된다.

3.2.5 wait-connection

- wait-connection [default : true] : 커넥션 풀에 가용한 매니지드 커넥션이 없을 경우 다른 클라이언트에 의해 반납되는 커넥션을 기다릴 것인지를 설정한다. 대기 풀에 들어온 클라이언트들은 들어온 순서대로 커넥션을 받게 된다.

- max-waiter-count [default : 5] : 커넥션을 얻기 위해 기다리는 클라이언트 수를 의미한다. 대기 풀의 크기이다.

- max-wait-trial [default : 5] : 대기풀 내에서 matchManagedConnection 을 시도하는 최대 횟수를 의미한다. 이를 넘은 클라이언트 요청은 바로 실패로 간주되어 리턴된다.

- wait-timeout [default : 10000] : 반납을 기다리는 최대 시간을 설정한다. 이를 넘은 클라이언트 요청은 실패로 간주되어 리턴된다.

3.2.6 disposable-connection

- allow-disposable-connection [default : true] : 커넥션을 얻지 못햇을 경우에 임시 커넥션을 만들것인지를 설정한다.

- max-disposable-count [default : 3] : 최대 임시 커넥션의 수이다.

4 Connection Management

4.1 소개

이번 절에서는 JEUS 와 Resource Adapter 간의 Connection Management Contract 에 대해 알아본다.

Connection Management Contract 는 다음과 같은 기능들을 제공한다.

- Managed 어플리케이션과 Non-managed (2-tier) 어플리케이션 모두에 대해 일관된 방법으로 Connection 을 얻을 수 있는 프로그래밍 모델을 제공한다.
- Resource Adapter 와 EIS 의 종류에 따라 CCI 에 기반한 Connection Factory 와 Connection 을 제공할 수 있다.
- 어플리케이션 서버가 다른 서비스들 – 트랜잭션, 보안, 풀링, 에러 추적/로깅 – 을 제공하는데에 일반적인 매커니즘을 제공한다.
- Connection Pooling 을 지원한다.

4.2 ConnectionFactory 와 Connection

ConnectionFactory 는 EIS 에 대한 Connection 을 제공해준다. Connector 는 CCI 혹은 EIS 특정적인 API 모두에 대해 일관된 프로그래밍 모델을 제공하는 것이 목표다. 이러한 목표를 위해 가이드라인을 제시하고 있다. CCI 의 경우와 Non-CCI 의 경우를 살펴보고 권장되는 패턴을 알아보도록 한다.

CCI ConnectionFactory 와 Connection 의 경우 아래 코드 샘플과 같다.

```
public inteface javax.resource.cci.ConnectionFactory extends
                java.io.Serializable, javax.resource.Referenceable
                {
    public javax.resource.cci.Connection getConnection()
                        throws javax.resource.ResourceException;
    …
}
```

```
public interface javax.resource.cci.Connection {
    public void close() throws javax.resource.ResourceException;
    …
}
```

Non-CCI ConnectionFactory 와 Connection 에는 JDBC interface 와 같은 경우가 해당될 수 있다. Non-CCI 의 경우 아래와 같은 코드 샘플을 갖는다.

```
public interface com.myeis.ConnectionFactory extends
   java.io.Serializable, javax.resource.Referenceable {
   public com.myeis.Connection getConnection()
      throws com.myeis.ResourceException;
   ...
}

public interface com.myeis.Connection {
   public void close() throws com.myeis.ResourceException;
   ...
}
```

인터페이스에 대해 다음과 같이 작성하도록 권장되고 있다.

- Resource Adapter 는 Connection 을 얻는데 있어 그 Connection Inteface 에서 기본적으로 정의되어 있는 것 이상의 추가적인 접근이 필요할 경우엔 getConnection 메소드들을 추가할 수 있다.
- Connection 인터페이스는 close 메소드를 제공해야 한다.

4.3 어플리케이션 프로그래밍 모델

EIS Connection 을 가져오는 방법에 대해 Managed 의 경우와 Non-managed 경우 각각에 대해 알아보도록 한다.

4.3.1 Managed Application Scenario

어플리케이션은 Lookup 을 통해 ConnectionFactory 를 가져오고 그 ConnectionFactory 를 통해 EIS Connection 을 가져오게 된다. 다음은 실제 어클리케이션이 Managed 환경에서 Connection 을 가져오는 과정을 설명한다.

1. 어플리케이션 어셈블러나 컴포넌트 제공자는 Deployment Descriptor 를 사용하여 Connection Factory 를 지정한다. 예를 들어

ResourceAdapter 의 Connection Factory 의 .이름이 eis/myEIS 이고 타입이 javax.resource.cci.ConnectionFactory 라 하면 EJB 의 Deployment Descriptor 는 아래와 같다.

<<*ejb-jar.xml*>>

```
<ejb-jar>
   . . .
   <enterprise-beans>
       . . .
       <session>
           . . .
           <resource-ref>
               <res-ref-name>eis/MyEIS</res-ref-name>
               <res-type>
                   javax.resource.cci.ConnectionFactory
               </res-type>
               <res-auth>Container</res-auth>
           </resource-ref>
           . . .
```

<<*jeus-ejb-dd_<modulename>.xml*>>

```
<jeus-ejb-dd>
    . . .
    <beanlist>
        <jeus-bean>
            . . .
            <res-ref>
                <jndi-info>
                    <ref-name>eis/MyEIS</ref-name>
                    <export-name>MYEISCONNECTOR</export-name>
                </jndi-info>
            </res-ref>
          . . .
```

2. Resource Adpater 의 세부 정보들을 Deploy 과정에서 설정한다. 어플리케이션 서버는 설정된 Resource Adapter 를 사용하여 EIS 와 물리적인 연결을 생성해낸다.

3. 어플리케이션 컴포넌트들은 JNDI 인터페이스를 이용하여 Connection Factory 를 가져올 수 있다.

```
// obtain the initial JNDI Naming context
   Context initctx = new InitialContext();
// perform JNDI lookup to obtain the connection factory
   javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)
        initctx.lookup("java:comp/env/eis/MyEIS");
```

NamingContext.lookup 에 전달된 JNDI 이름은 res-ref-name 에 설정된 값과 일치한다. JNDI lookup 결과는 res-type 에서 명시한 바와 같이 javax.resource.cci.ConnectionFactory 타입의 인스턴스가 반환된다.

4. 어플리케이션 컴포넌트는 getConnection 메소드를 사용하여 Connection Factory 로부터 EIS Connection 을 얻어 올 수 있다. 반환된 Connection 은 실제 물리적인 Connection 에 대한 어플리케이션 레벨의 핸들을 나타낸다. 여러 Connection 을 얻기 위해 getConnection 메소드를 여러 번 호출할 수도 있다.

   ```
   javax.resource.cci.Connection cx= cxf.getConnection();
   ```

5. 어플리케이션 컴포넌트는 반환된 Connection 을 사용하여 Resource Adapter 를 통해 EIS 에 접근할 수 있다.
6. 컴포넌트가 Connection 의 사용을 끝냈을 때 Connection 인터페이스의 close 메소드를 사용해 Connection 을 close 해야 한다.
   cx.close();
7. 어플리케이션 컴포넌트가 할당된 Connection 을 close 하는데 실패한 경우 그 Connection 은 사용하지 않는 Connection 으로 분류된다. 어플리케이션 서버가 그 사용하지 않는 Connection 들을 정리하게 된다. 컴포넌트 인스턴스가 컨테이너에서 종료되는 경우 컨테이너는 그 컴포넌트 인스턴스에 의해 사용된 모든 Connection 을 정리한다.

4.3.2 Non-managed Application Scenario

J2EE WAS 의 관여 없이 직접 Connector 를 사용하는 것도 가능하다. 이 같은 모델을 “Non-managed”라고 부른다. 아래에 예제 코드가 있다.

```
...
// EJB code:
// First create an instance of a ManagedConnectionFactory
// implementation class, passing in initialization parameters
// (if any) for this instance
   com.myeis.ManagedConnectionFactoryImpl mcf =
```

```
          new com.myeis.ManagedConnectionFactoryImpl(...);
// Set properties on the ManagedConnectionFactory instance
// 참고: Properties are defined on the implementation class and
// not on the javax.resource.spi.ManagedConnectionFactory
// interface.
mcf.setServerName(...);
mcf.setPortNumber(...);
// ... set remaining properties
// Get access to connection factory. The ConnectionFactory
// instance gets initialized with the default ConnectionManager
// provided by the resource adapter:
   javax.resource.cci.ConnectionFactory cxf =
          (javax.resource.cci.ConnectionFactory)
    mcf.createConnectionFactory();
// Get a connection using the ConnectionFactory instance
   javax.resource.cci.Connection cx = cxf.getConnection(...);
// ... use connection to access the underlying EIS instance
// Close the connection
   cx.close();
...
```

Managed Connection Factory 의 프로퍼티가 커넥션을 요청하기 전에 어떻게 설정되는지 주목하기 바란다(첫 번째와 두 번째 굵은 줄). Managed scenario 에서는 이런 프로퍼티가 ra.xml 에서 제공되어야 한다.

4.4 Connection Example

Outbound Resource Adapter 를 사용하여 하나 혹은 그 이상의 Outbound Connection 을 설정할 수 있다. 다음은 javax.sql.DataSource 를 Connection Factory 로 가지고 java.sql.Connection 을 Connection 으로 갖는 Connection 설정의 예이다.

<<ra.xml>>

```
<?xml version="1.0" encoding="UTF-8"?>
<connector xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.5"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/connector_1_5.xsd">
```

```
<display-name>ConnectionManagementSample1</display-name>
<vendor-name>Tmax Soft</vendor-name>
<eis-type>My Data</eis-type>
<resourceadapter-version>1.0</resourceadapter-version>
<license>
    <license-required>false</license-required>
</license>
<resourceadapter>
    <outbound-resourceadapter>
        <connection-definition>
            <managedconnectionfactory-class>
                cm.MyManagedConnectionFactory
            </managedconnectionfactory-class>
            <connectionfactory-interface>
                javax.sql.DataSource
            </connectionfactory-interface>
            <connectionfactory-impl-class>
                cm.MyDataSource
            </connectionfactory-impl-class>
            <connection-interface>
                java.sql.Connection
            </connection-interface>
            <connection-impl-class>
                cm.MyConnection
            </connection-impl-class>
        </connection-definition>
        <transaction-support>
                NoTransaction
        </transaction-support>
        <authentication-mechanism>
            <authentication-mechanism-type>
                 BasicPassword
             </authentication-mechanism-type>
            <credential-interface>
              javax.resource.spi.security.PasswordCredential
             </credential-interface>
        </authentication-mechanism>
        <reauthentication-support>
              false
```

```
          </reauthentication-support>
       </outbound-resourceadapter>
    </resourceadapter>
</connector>
```

<<jeus-connector-dd.xml>>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jeus-connector-dd xmlns="http://www.tmaxsoft.com/xml/ns/jeus">
    <module-name>cm1.rar</module-name>
    <use-workmanager>false</use-workmanager>
    <connection-pool>
        <export-name>MyConnector</export-name>
        <transaction-support>NoTransaction</transaction-support>
        <pool-management>
            <min>5</min>
            <max>20</max>
        </pool-management>
    </connection-pool>
</jeus-connector-dd>
```

5 Transaction Management

이번 절에서는 JEUS 와 EIS Resource Manager 간의 Transaction Contract 에 대해 알아본다. JEUS 서버에서 EIS 로 나가는 Outbound 통신의 경우와 EIS 로부터 JEUS 서버로 들어오는 Inbound 통신에 대한 시스템 레벨의 Transaction Management Contract 가 있다. 어플리케이션 레벨의 Trnasaction Management 에 대해선 EJB 와 같은 J2EE 컴포넌트 스펙을 살펴본다.

5.1 지원되는 Transaction 레벨

JEUS 서버는 다음과 같은 Transaction 레벨을 지원한다.

- XA Transaction – Transaction 이 Resource Adapter 외부의 Transaction Manager 에 의해 관리된다. Resource Adapater 는 ra.xml 파일에 transaction-support 엘리먼트에 지원할 Trnasaction 타입을 정의한다. 어플리케이션 컴포넌트가 EIS Connection 요청을 Transaction 의 일부로 지정할 경우 JEUS 는 XA Resource 를 Transaction Manager 에 등록한다. 어플리케이션 컴포넌트가 그 Connection 을 닫는 경우엔 JEUS 는 Transaction 이 종료되면 그 EIS Connection 을 정리한다.

- Local Transaction – 앞서 1 장에서 설명한 바와 같이 Transaction 에 하나의 Resource Manager 만 참여하고 있는 경우 Local Transaction 을 사용할 수 있다. XA Transaction 과 달리 Local Transaction 은 2-Phase Commit 프로토콜에 참여하지 않는다.

- No Transaction – Resource Adapter 가 No Transaction 으로 설정된 경우에도 Resource Adapter 는 Transaction Context 안에서 사용될 수 있다. 하지만 이 경우 그 Resource Adapter 에 사용되는 Connection 들은 Transaction 에 등록되지 않고 마치 Transaction 이 없는 거 처럼 동작한다. 즉 이 Connection 을 사용한 Operation 들은 즉시 실행되고 만약 Transaction 이 Roll-Back 되더라도 그 Operation 들에 의한 변화는 복원되지 않는다.

5.2 Tranaction 레벨 설정

Resource Adapter 에 대한 Transaction 레벨 설정은 ra.xml Deployment Descriptor 파일에 할 수 있다.

- XATransaction : XA 트랜잭션과 로컬 트랜잭션을 모두 지원한다.
- LocalTransaction : 로컬 트랜잭션만을 지원하는 커넥션이다.
- NoTransaction : 트랜잭션을 지원하지 않는 커넥션이다.

jeus-connector-dd.xml Deployment Descriptor 파일을 이용하여 ra.xml 의 설정을 덮어 쓸 수도 있다. jeus-connector-dd.xml 파일의 transaction-support 에 설정된 값은 ra.xml 파일에 설정된 값에 우선한다. 가능한 Transaction 레벨 설정은 다음과 같다. XATransactionOnly 가 추가되어 있다. XATransactionOnly 는 J2CA 표준은 아니며 XATransaction 만을 지원하고 Local Trasaction 최적화를 사용하지 않는다.

- NoTransaction
- LocalTransaction
- XATransaction
- XATransactionOnly

6 Security

이 절은 Resource Adapter 의 Securtity Contract 에 대해 알아본다.

6.1 Container-Managed 와 Application-Managed Sign-on

J2CA 1.5 스펙에 명시된 바와 같이 JEUS 서버는 Container-Managed 와 Application-Managed 인증 방식 모두 지원하고 있다. 런타임시 JEUS 는 호출한 클라이언트 컴포넌트의 Deployment Descriptor 에 기술된 내용을 바탕으로 Sign-on 메커니즘을 판별하게 된다.

6.1.1 Application-Managed Sign-on

Application-Managed Sign-on 의 경우 EIS 에 대한 Connection 을 얻기 위해 호출할 때 클라이언트 컴포넌트가 필요한 Security 정보(대개 Username 과 Password)를 넘겨주게 된다. 이 경우 Connection 을 요청할 때 Application Server 는 추가적인 Security 정보를 제공하지 않는다. Resource Adapter 는 클라이언트 컴포넌트가 제공한 Security 정보를 이용하여 EIS 에 Sign-on 을 수행한다.

6.1.2 Container-Managed Sign-on

Container-Managed Sign-on 을 사용하기 위해서는 JEUS 서버에 Resource Principal 을 명시하고 Connection 요청시 그 Resource Principal 을 사용해야 한다.

6.2 Authentication Mechanism

J2CA 1.5 스펙에 따라 Credential 은 javax.security.auth.Subject 에 저장된다. Credential 은 ManagedConnectionFactory 에서 createManagedConnection()이나 matchManagedConnection() 메소드를 호출할 때 전달된다.

Password Authentication 은 User ID 와 Password 로 구성된다. User 가 Resource Adapter 에 Connection 을 요청할 경우 적절한 Credential 이 그 Resource Adapter 에 전달된다.

SSL 프로토콜이 Password Authentication 에 추가적으로 적용될 수 있다.

6.3 Security Identity 설정

Security 인증 방식이 Container-Managed 인 경우 jeus-connector-dd.xml 파일에 user, password 값을 설정할 수 있다. 여기에 설정된 값은 ra.xml 에 설정된 값보다 우선시 된다.

A. jeus-connector-dd.xml XML Configuration Reference

A.1 소개

본 부록의 레퍼런스는 jeus-connector-dd.xml(핵심 컨테이너 구성파일)의 모든 XML 태그에 대해서 설명하고 있다. 이 파일의 XSD 파일은 “JEUS_HOME\config\XSDs\”디렉토리의 “jeus-connector-dd.XSD”파일이다.

본 레퍼런스는 3 부분으로 나눠져있다.

8. **XML Schema/XML 트리**: XML 설정 파일의 모든 태그 리스트를 정리했다. 각 노드의 형식은 다음과 같다.

    a. 태그 레퍼런스로 빨리 찾아보기 위해서 각 태그마다 인덱스 번호(예 (11))를 붙여놓았다. 태그 레퍼런스에서는 이 번호 순서로 설명한다.

    b. XML Schema 에서 정의한 XML 태그명을 <**tag name**> 형식으로 표시한다.

    c. XML Schema 에서 정의한 Cardinality 를 표시한다. “?” = 0 개나 1 개의 element, “+” = 1 개 이상의 element, “*” = 0 개 이상의 element, (기호가 없음) = 정확히 1 개의 element

    d. 몇몇 태그에는 “P” 문자를 붙여놓았는데, 해당 태그는 성능에 관계되는 태그라는 것을 뜻한다. 이 태그는 설정을 튜닝할 때 사용된다.

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.

A.2 XML Schema/XML 트리

```
(1) <jeus-connector-dd>
      (2) <module-name>
      (3) <use-workmanager>? P
      (4) <worker-pool>?
            (5) <min>? P
            (6) <max>? P
            (7) <step>? P
            (8) <pre-allocation>? P
            (9) <submit-timeout>? P
            (10) <release-timeout>? P
            (11) <shutdown-timeout>? P
            (12) <pooled-timeout>? P
      (13) <connection-pool>*
            (14) <export-name>?
            (15) <transaction-support>?
            (16) <user>?
            (17) <password>?
            (18) <use-wrapper>? P
            (19) <dissociation-timeout>? P
            (20) <invalidation-timeout>? P
            (21) <skip-connection-matching>? P
            (22) <pool-management>?
                  (23) <min>? P
                  (24) <max>? P
                  (25) <pooled-timeout>? P
                  (26) <wait-connection>?
                        (27) <wait-connection>? P
                        (28) <max-waiter-count>? P
```

```
                (29) <max-wait-trial>? P
                (30) <wait-timeout>? P
          (31) <disposable-connection>?
                (32) <allow-disposable-connection>? P
                (33) <max-disposable-count>? P
    (34) <property>*
          (35) <name>
          (36) <type>
          (37) <value>
```

A.3 Element Reference

(1) <jeus-connector-dd>

Description	단일 JEUS Connector 모듈의 최상위 element. 각각의 jeus-connector-dd.xml 파일에는 이 태그가 반드시 존재한다.
Child Elements	(2)module-name (3)use-workmanager? (4)worker-pool? (13)connection-pool*

(2) <jeus-connector-dd> <module-name>

Description	Connector 모듈의 이름이다. 어플리케이션 내의 모듈인 경우 어플리케이션 내에서 유일하여야 하며 서버 모듈인 경우는 전체 JEUS 시스템에서 유일하여야 한다. Message Driven Bean 에서 inflow 타겟을 지정하기 위해 사용된다.
Value Type	token
Example	<module-name>whitebox-xa</module-name>

(3) <jeus-connector-dd> <use-workmanager>

Description	Work Manager 를 사용할 것인지를 설정한다. true 인 경우 리소스 어댑터가 시작(start) 되기 직전에 설정에 따라 초기화 된다.
Value Type	boolean
Default Value	false
Example	<use-workmanager>true</use-workmanager>

(4) <jeus-connector-dd> <worker-pool>

Description	Work Manager 에 관련된 설정을 한다.
Child Elements	(5)min? (6)max? (7)step?

```
(8)pre-allocation?
(9)submit-timeout?
(10)release-timeout?
(11)shutdown-timeout?
(12)pooled-timeout?
```

(5) <jeus-connector-dd> <worker-pool> <min>

Description	Work Manager 가 관리하는 쓰레드 갯수의 최소값이다. 이 설정값 보다 적은 쓰레드가 존재할 경우 Work Manager 는 사용자의 요청에 따라 즉시 쓰레드를 늘인다. 이 설정값 보다 많은 쓰레드가 존재하는 경우 일정 시간동안 사용되지 않은 쓰레드를 줄이기 시작하여 min 갯수의 쓰레즈를 유지한다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	3
Example	`<min>3</min>`

(6) <jeus-connector-dd> <worker-pool> <max>

Description	Work Manager 가 관리하는 쓰레드 갯수의 최대값이다. 쓰레드의 갯수는 이 값을 넘지 않는다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	10
Example	`<max>10</max>`

(7) <jeus-connector-dd> <worker-pool> <step>

Description	쓰레드 수가 증가할 필요가 있을 경우, 한번에 증가하게 되는 쓰레드의 갯수이다. 총 쓰레드의 수는 위에 설정된 max 값을 넘을 수 없다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	1
Example	`<max>1</max>`

(8) <jeus-connector-dd> <worker-pool> <pre-allocation>

Description	Work Manager 가 초기화 될때 min 값에 설정된 수의 쓰레드를 미리 만들어 놓는다.
Value Type	boolean
Default Value	true

Example	`<pre-allocation>true</pre-allocation>`

(9) <jeus-connector-dd> <worker-pool> **<submit-timeout>**

Description	클라이언트의 요청에 대해 지정된 시간동안 작업을 시작할 수 없을 경우 작업은 거부된다. 사용자는 서비스 요청 방식에 따라 익셉션을 받거나 이벤트를 받게 된다.
Value Description	시간값 msec 단위. 0 으로 설정하는 경우 작업이 시작될 때까지 계속 대기상태로 기다린다.
Value Type	nonNegativeLongType
Value Type Description	0 이상의 long type 이다. 즉, long 범위에서 0 이상의 값들을 포함한다.
Default Value	5000
Example	`<submit-timeout>5000</submit-timeout>`

(10) <jeus-connector-dd> <worker-pool> **<release-timeout>**

Description	작업이 시작되어 이 시간이 지난 후에도 작업이 종료되지 않았다면 Work Manager 는 해당 작업에 대해 release() 메쏘드를 호출해 준다.
Value Description	시간값 msec 단위. 0 으로 설정하는 경우 release() 메쏘드는 호출되지 않는다.
Value Type	nonNegativeLongType
Value Type Description	0 이상의 long type 이다. 즉, long 범위에서 0 이상의 값들을 포함한다.
Default Value	60000
Example	`<release-timeout>60000</release-timeout>`

(11) <jeus-connector-dd> <worker-pool> **<shutdown-timeout>**

Description	Work Manager 가 종료되는 순간 진행중인 작업이 있을 경우 최대 여기에 설정된 시간동안 대기한다.
Value Description	시간값 msec 단위. 0 이면 기다리지 않으며, -1 로 설정하는 경우 작업이 종료될 때까지 계속 기다리게 된다. 작업중인 쓰레드들은 Interrupt 를 받게 된다.
Value Type	long
Default Value	-1
Example	`<shutdown-timeout>-1</shutdown-timeout>`

(12) <jeus-connector-dd> <worker-pool> **<pooled-timeout>**

Description	Work Manager 에 min 값을 초과하는 수의 쓰레드가 있을 경우 설정된 시간 동안 사용되지 않은 쓰레드를 풀에서 제거한다.
Value Description	시간값 msec 단위. 0 이면 제거하지 않는다.
Value Type	nonNegativeLongType

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

Default Value 600000

Example
```
<pooled-timeout>600000</pooled-timeout>
```

(13) <jeus-connector-dd> **<connection-pool>**

Description Connection Pool 에 대한 설정을 한다.

Child Elements
```
(14)export-name?
(15)transaction-support?
(16)user?
(17)password?
(18)use-wrapper?
(19)dissociation-timeout?
(20)invalidation-timeout?
(21)skip-connection-matching?
(22)pool-management?
(34)property*
```

(14) <jeus-connector-dd> <connection-pool> **<export-name>**

Description 해당 connection pool 이 JNDI Naming System 에 바인딩 되는 이름이다.

Value Description 임의로 지정할 수 있고 해당 모듈내에서 유일한 이름이어야만 한다.

Value Type token

Example
```
<export-name>datasource1</export-name>
```

(15) <jeus-connector-dd> <connection-pool> **<transaction-support>**

Description 해당 connection pool 이 지원하는 트랜잭션 타입을 설정한다. 여기에 설정된 값은 ra.xml 에 설정된 값보다 우선시 된다.

Value Description NoTransaction, LocalTransaction, XATransaction, XATransactionOnly 네가지 중 하나를 설정한다.

Value Type transaction-supportType

Defined Value NoTransaction

LocalTransaction

XATransaction

XATransactionOnly

Example
```
<transaction-support>XATransaction</transaction-
support>
```

(16) <jeus-connector-dd> <connection-pool> **<user>**

Description security 를 container 에서 관리하는 경우, connection 을 생성하기 위해 사용하는 user 이름이다.

Value Type	token
Example	`<user>scott</user>`

`(17) <jeus-connector-dd> <connection-pool>` **`<password>`**

Description	security 를 container 에서 관리하는 경우, connection 을 생성하기 위해 사용하는 password 이다.
Value Type	token
Example	`<password>tiger</password>`

`(18) <jeus-connector-dd> <connection-pool>` **`<use-wrapper>`**

Description	부가적인 기능을 위해서 connection wrapper 를 이용한다. 자세한 내용은 메뉴얼을 참조하도록 한다.
Value Type	boolean
Default Value	true
Example	`<use-wrapper>true</use-wrapper>`

`(19) <jeus-connector-dd> <connection-pool>` **`<dissociation-timeout>`**

Description	dissociation 을 적용하는 시간 값을 설정한다. 해당 리소스 어댑터에서 이 기능을 지원하지 않는 경우 무시된다.
Value Description	시간값 msec 단위. 0 이면 dissociation 을 사용하지 않는다.
Value Type	nonNegativeLongType
Value Type Description	0 이상의 long type 이다. 즉, long 범위에서 0 이상의 값들을 포함한다.
Default Value	0
Example	`<dissociation-timeout>0</dissociation-timeout>`

`(20) <jeus-connector-dd> <connection-pool>` **`<invalidation-timeout>`**

Description	invalidation 을 적용할 시간 값을 설정한다. 이 시간이 지나도 connnection pool 로 돌아오지 않는 connection 은 강제로 제거 된다.
Value Description	시간값 msec 단위. 0 이면 invalidation 을 사용하지 않는다.
Value Type	nonNegativeLongType
Value Type Description	0 이상의 long type 이다. 즉, long 범위에서 0 이상의 값들을 포함한다.
Default Value	0
Example	`<invalidation-timeout>0</invalidation-timeout>`

`(21) <jeus-connector-dd> <connection-pool>` **`<skip-connection-matching>`**

Description	connection 인증을 위한 connection matching 과정을 생략한다.

Value Type	boolean
Default Value	false
Example	`<skip-connection-matching>false</skip-connection-matching>`

`(22) <jeus-connector-dd> <connection-pool> <pool-management>`

Description	connection pool 에 관련된 값을 설정한다.

Child Elements	`(23)min?`
	`(24)max?`
	`(25)pooled-timeout?`
	`(26)wait-connection?`
	`(31)disposable-connection?`

`(23) <jeus-connector-dd> <connection-pool> <pool-management> <min>`

Description	Connection pool 에 의해 관리되는 Connection 갯수의 최소값이다. 이 설정값 보다 적은 connection 이 존재할 경우 사용자의 요청에 따라 즉시 새로운 connection 을 생성한다. 이보다 많은 connection 이 존재하는 경우 일정 시간동안 사용되지 않은 connection 를 풀에서 제거한다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	0
Example	`<min>0</min>`

`(24) <jeus-connector-dd> <connection-pool> <pool-management> <max>`

Description	Connection pool 에 의해 관리되는 Connection 갯수의 최대값이다.
Value Type	nonNegativeIntType
Value Type Description	0 이상의 int type 이다. 즉, int 범위에서 0 이상의 값들을 포함한다.
Default Value	20
Example	`<max>20</max>`

`(25) <jeus-connector-dd> <connection-pool> <pool-management> <pooled-timeout>`

Description	Connection pool 에 min 값보다 많은 Connection 이 있을 경우, 설정된 시간 동안 사용되지 않은 Connection 을 풀에서 제거한다
Value Description	시간값 msec 단위. 0 으로 설정하면 connection 은 제거되지 않는다.
Value Type	nonNegativeLongType
Value Type Description	0 이상의 long type 이다. 즉, long 범위에서 0 이상의 값들을 포함한다.
Default Value	600000

Example `<pooled-timeout>600000</pooled-timeout>`

(26) <jeus-connector-dd> <connection-pool> <pool-management> **<wait-connection>**

Description Connection Pool 에서 Connection 을 얻기 위해 기다릴 필요가 있을 경우 이에 관련된 설정이다.

Child Elements
```
(27)wait-connection?
(28)max-waiter-count?
(29)max-wait-trial?
(30)wait-timeout?
```

(27) <jeus-connector-dd> <connection-pool> <pool-management> <wait-connection> **<wait-connection>**

Description Connection 을 얻기 위해 기다릴 것인지를 설정한다.

Value Type boolean

Default Value true

Example `<wait-connection>true</wait-connection>`

(28) <jeus-connector-dd> <connection-pool> <pool-management> <wait-connection> **<max-waiter-count>**

Description 커넥션을 얻기 위해 기다리는 client 의 최대 수이다. 이보다 많을 경우 바로 실패한 것으로 간주된다.

Value Type nonNegativeIntType

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

Default Value 5

Example `<max-waiter-count>5</max-waiter-count>`

(29) <jeus-connector-dd> <connection-pool> <pool-management> <wait-connection> **<max-wait-trial>**

Description 이 횟수만큼 connection matching 을 실패했다면 Connection 을 얻는데 실패한 것으로 간주한다.

Value Type nonNegativeIntType

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

Default Value 5

Example `<max-waiter-trial>5</max-waiter-trial>`

(30) <jeus-connector-dd> <connection-pool> <pool-management> <wait-connection> **<wait-timeout>**

Description 이 시간동안 기다려도 Connection 을 얻지 못했다면 Connection 을 얻는데 실패한 것으로 간주한다.

Value Type nonNegativeLongType

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

Default Value 10000

Example
```
<wait-timeout>10000</wait-timeout>
```

(31) <jeus-connector-dd> <connection-pool> <pool-management> **<disposable-connection>**

Description Connection 을 얻지 못했을 경우 임시 Connection 을 이용할 것인지에 대한 설정이다.

Child Elements
```
(32)allow-disposable-connection?
(33)max-disposable-count?
```

(32) <jeus-connector-dd> <connection-pool> <pool-management> <disposable-connection> **<allow-disposable-connection>**

Description Connection 을 얻지 못했을 경우 임시 Connection 을 사용할 것인지를 설정한다.

Value Type boolean

Default Value true

Example
```
<allow-disposable-connection>true</allow-
disposable-connection>
```

(33) <jeus-connector-dd> <connection-pool> <pool-management> <disposable-connection> **<max-disposable-count>**

Description 임시 Connection 의 최대 갯수이다. 임시 커넥션은 사용이 끝나면 자동적으로 제거된다.

Value Type nonNegativeIntType

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

Default Value 3

Example
```
<max-disposable-count>3</max-disposable-count>
```

(34) <jeus-connector-dd> <connection-pool> **<property>**

Description ManagedConnectionFactory 에 적용할 property 를 추가한다. ra.xml 에 설정된 값보다 우선시 된다.

Child Elements
```
(35)name
(36)type
(37)value
```

(35) <jeus-connector-dd> <connection-pool> <property> **<name>**

Value Type token

(36) <jeus-connector-dd> <connection-pool> <property> **<type>**

Value Type token

```
(37) <jeus-connector-dd> <connection-pool> <property> <value>
```

Value Type token

색인