# <JSTORM>

## Enterprise Java Beans 5부

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

## Document Information

| | |
|---|---|
| Document title: | Enterprise Java Beans 5부 |
| Document file name: | EJB5_JSTORM.doc |
| Revision number: | <0.9> |
| Issued by: | <박지훈> pjh@jstorm.pe.kr |
| | <남궁윤> yoonng@hanmail.net |
| | <김종윤> foxkjy@chollian.net |
| | <전부현> kanaloa@orgio.net |
| | 정리 : <윤준호> junoyoon@orgio.net |
| Issue Date: | <2000/9/7> |
| Status: | final |

## Content Information

| | |
|---|---|
| Audience | EJB 프로그래밍을 학습하는 중급 JAVA 프로그래머 |
| Abstract | 트랜잭션과 디플로이먼트에 대하여 알아본다. |
| Reference | 1) Enterprise Java Bean, Second Edition (Oreill'y) |
| | 2) Mastering EJB (Wiley) |
| | 3) http://www.itplus.co.kr 자료 |
| | 4) http://www.bea.com의 자료 |
| | 5) http://www.javasoft.com의 자료 |
| | 6) http://www.javaworld.com의 자료 |
| | 7) Sun사의 EJB 스펙 1.1 PDF파일 |
| | 8) J2EE SDK의 example9 |
| | 9) J2EE blueprint 및 java pet store example |
| Benchmark information | |

## Document Approvals

| | | |
|---|---|---|
| 고문 | Signature | date |
| |  | |
| 지위 | Signature | date |

## Revision History

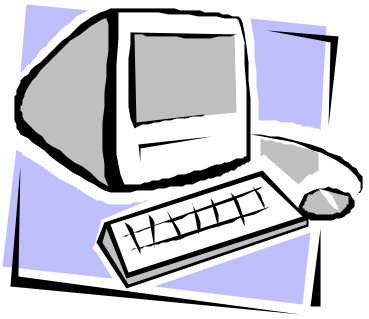
| Revision | Date | Author | Description of change |
|---|---|---|---|
| | | | |

# Table of Contents

# 5 부

# 마무리! 트랜잭션, 디플로이먼트

지금까지 EJB를 이용한 프로그래밍을 다 살펴보았다. 지금까지 다 따라왔으면 당신은 EJB의 전문가가 되었을 것이다. 이번 장에서는 마무리로 트랜잭션과 디플로이먼트에 대해 알아보도록 하자. 얼마 남지 않았으니까 열심히 따라오기 바란다.



# Chapter 12

# Transaction Management

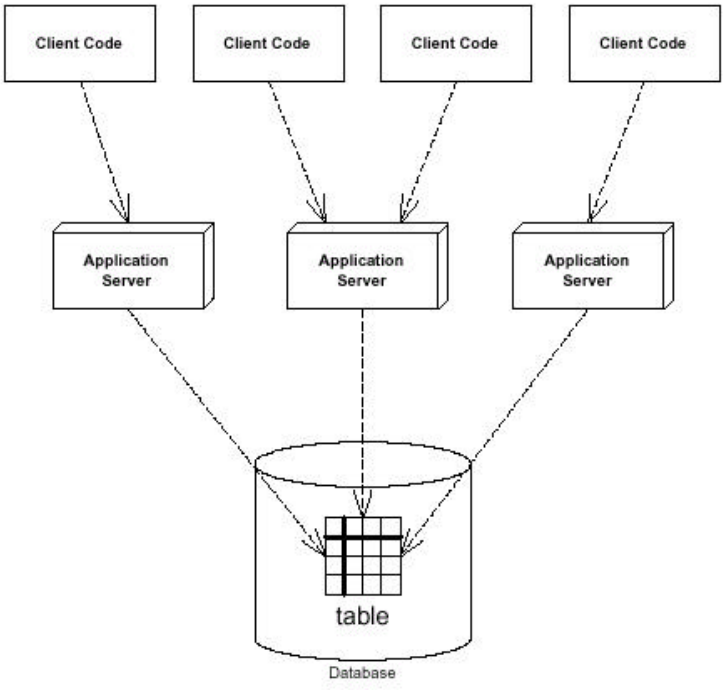
## ACID 트랜잭션

### Transaction이란?

✍ 동기

All or Nothing !

은행에서 예금계좌에서 이체할 때 서버가 다운이 되면?




네트웍/기계의 다운 And/Or 여러 사용자가 동시에 공용 데이터영역을 사용

-> ACID Transaction이 필요

## ACID 속성

✍ Atomicity

- all or nothing

✍ Consistency

- 트랜잭션이 끝난 후 데이터들은 일관성을 유지해야 함.

✍ Isolation

- 여러 트랜잭션이 동시에 수행할 때 한 트랜잭션이 다른 트랜잭션 수행중인 불완전한 데이터를 사용하는것을 방지함.

- Serialization 제공

✍ Durability

- 트랜잭션 수행결과는 반드시 Secondary Storage에 반영이 되어 있어야 한다.

## Transactional Model

✍ Flat Transaction

a. 가장 단순한 트랜잭션 모델

b. Atomic한 " Single Unit of Work "의 오퍼레이션들의 연속 집합

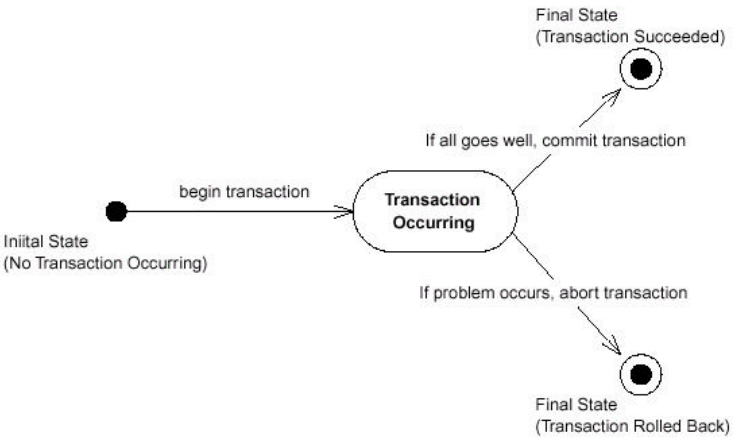
c. EJB 1.1이 지원하는 트랜잭션 모델

d. Vocabulary

Commit : 트랜잭션이 성공적으로 완료되어 스토리지에 반영됨.

Roll Back : 트랜잭션이 실패하여 트랜잭션이 시작되기 이전의 상태로 돌려놓음.

e. 상태 다이어그램




<그림> 플랫 트랜잭션의 상태 변이

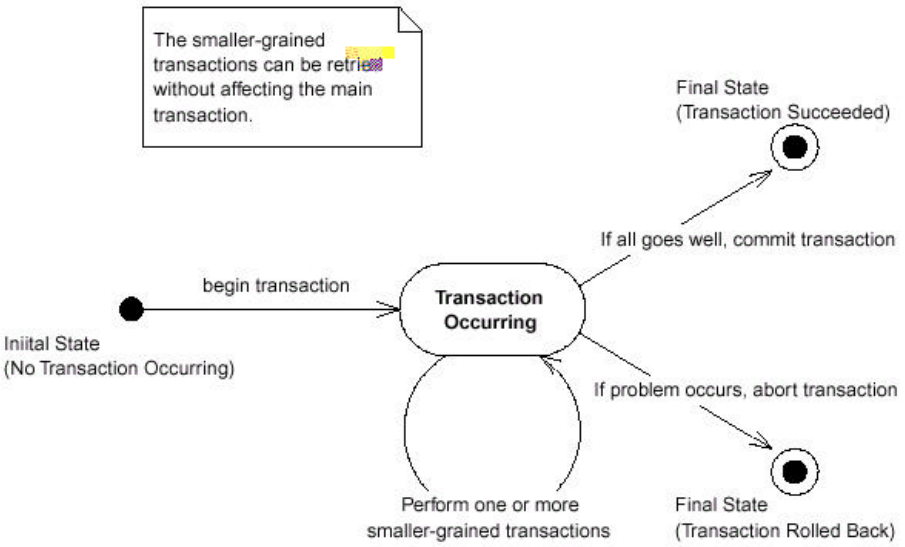
✍ Nested Transaction

a. 한 작업 단위 안에 " Atomic Units of Work " 가 포함될 수 있다. 따라서 포함되어 있는 서브 트랜잭션은 전체 트랜잭션을 RollBack하지 않고 자신만 RollBack할 수 있다.

b. Main/Sub Transaction

c. EJB 1.1에서 지원을 필수조건으로 하지 않음.

d. 상태 다이어그램

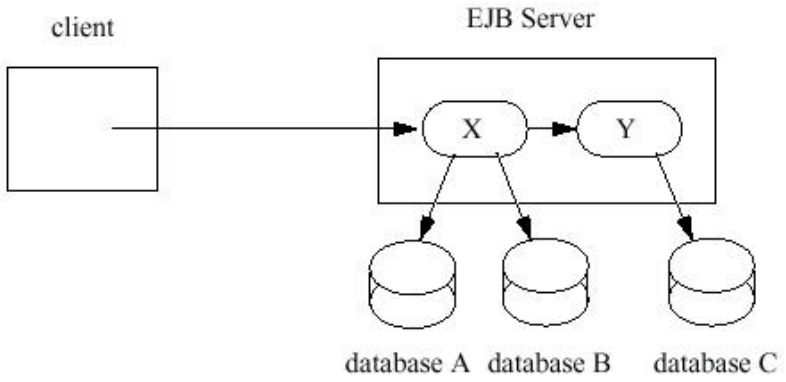



<그림> *Nested Transaction*의 상태변이

# EJB 트랜잭션 시나리오

## 여러 개의 데이터베이스의 업데이트

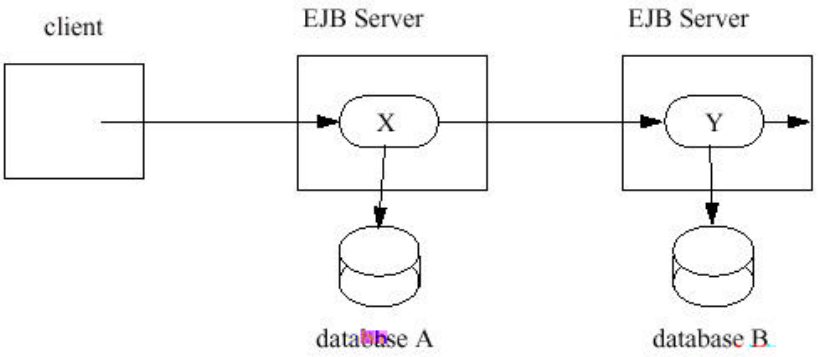
- 한 어플리케이션 프로그램이 한 트랜잭션내에서 여러 데이터베이스에 있는 데이터들을 업데이트할 때 트랜잭션이 보장된다.
- 사례




<그림> 여러 데이터베이스들과 연결된 트랜잭션

## 여러 개의 EJB 서버들을 통한 데이터베이스들의 업데이트

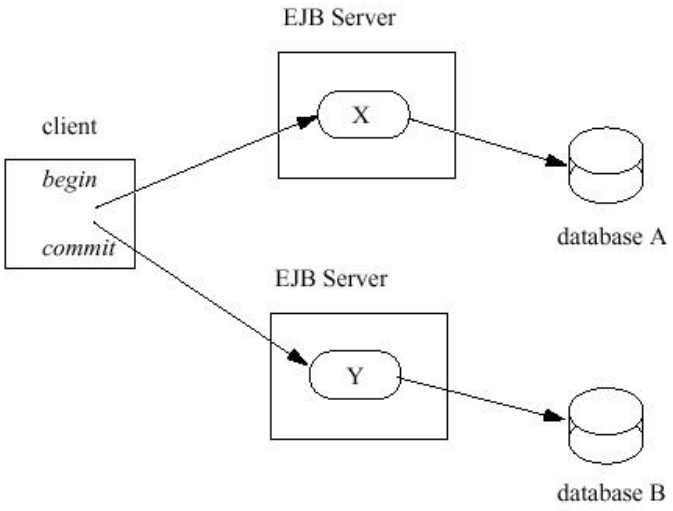
- 한 트랜잭션내에서 EJB서버가 설치된 여러 개의 사이트들의 데이터베이스들을 업데이트할 수 있다.
- 사례




<그림> 분산된 EJB서버들에 걸친 트랜잭션 처리

## 클라이언트 소스코드내에서 트랜잭션을 시작하며 관리

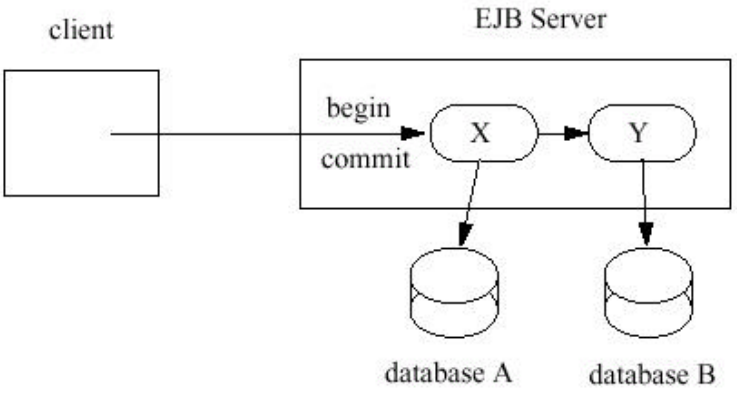
- javax.transaction.UserTransaction 인터페이스를 사용하여 트랜잭션을 시작, 관리할 수 있음.
- J2EE의 JNDI API를 사용하여 UserTransaction 객체를 확보할 수 있음.
- 트랜잭션 전문가만 사용하도록 권고
- 사례




<그림> 클라이어트가 시작한 트랜잭션

## EJB 컨테이너에 의해 트랜잭션이 자동으로 시작되며 관리

- EJB 빈 소스코드내에 트랜잭션 코드가 포함되지 않고 6개의 트랜잭션 속성을 Deploy할 때 지정함.
- 클라이언트내에 트랜잭션 코드가 없어도 자동으로 EJB 컨테이너가 실행함.
- 사례



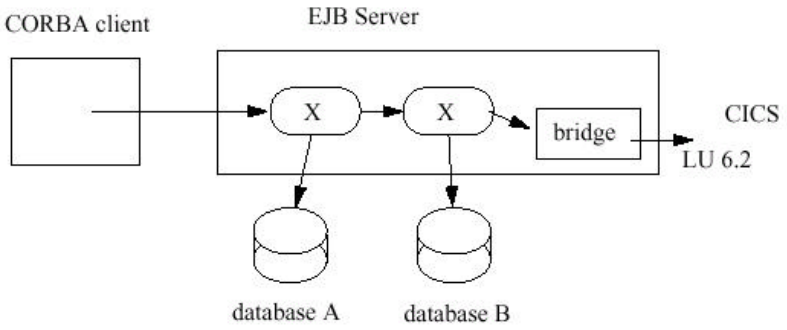

<그림> EJB 컨테이너가 자동으로 시작한 트랜잭션

## EJB 빈 소스코드내에서 트랜잭션을 시작하고 관리

- 세션빈내에서 javax.transaction.UserTransaction 을 사용하여 트랜잭션을 프로그래밍할 수 있음.

## 자바가 아닌 클라이언트 또는 서버와의 호환성

- CORBA가 실행가능한 EJB 서버내의 EJB 빈을 코바객체가 호출할 수 있음.
- 브릿지를 사용하여 EJB빈이 다른 서버내의 컴포넌트를 호출할 수 있음.
- 브릿지 및 CORBA 실행가능한 EJB서버는 EJB 스펙범위 밖임.
- 사례




<그림> 코바와 다른 시스템과 연동하는 EJB 컴포넌트

# EJB 트랜잭션 타입

## 개발자 입장의 트랜잭션 종류

✍✍ 트랜잭션은 일단 Begin 하면 Commit 이거나 Abort 이다. Abort 일 경우는 RollBack을 통해 트랜잭션이 시작하지 않은 상태가 된다.

✍✍ 트래잭션 영역(boundary)의 문제

WHO : 누가 트랜잭션을 시작하는가?

WHO : 누가 Commit 또는 Abort하라고 지시할 것인가?

When : 언제 위의 각 스텝이 일어나는가?

✍✍ EJB의 트랜잭션 영역

Programmatically : EJB 빈 소스 코드 내에 트랜잭션 코드 삽입

Declaratively : EJB 빈 소스코드는 전혀 건드리지 않고 트랜잭션 정보를 Deployment Descriptor에 명시

✍✍ "Programmatically VS Declaratively" EJB 트랜잭션 타입

- Programmatically

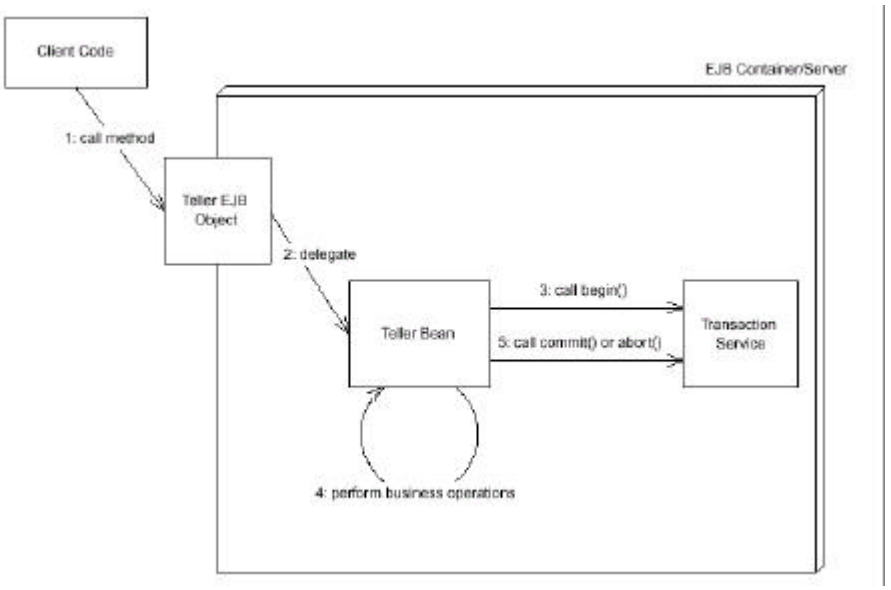
a. 책임

EJB 프로그래머가 트랜잭션과 Commit, Abort를 지시한다.

EJB 프로그래머가 소스코드내에 배치한 트랜잭션 코드에 따라 위의 과정이 발생한다.

b. JTA(Java Transaction API)를 사용하여 코딩한다.

c. Programmatical 한 EJB 트랜잭션 실행 과정




<그림> programmatic 트랜잭션

Declaratively

a. 책임

EJB Deployer가 Deployment Descriptor안에 트랜잭션 방식을 지시한다.

EJB 컨테이너가 트랜잭션의 시작과 Commit, Abort를 지시한다.

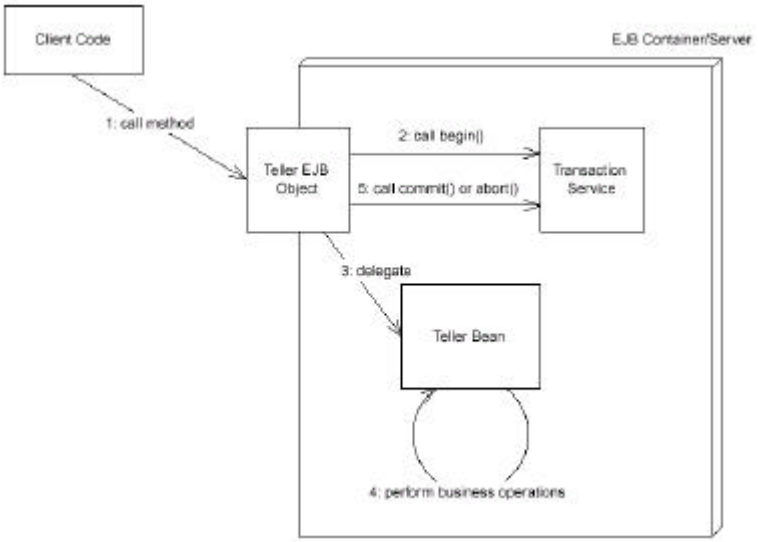
EJB 컨테이너의 트랜잭션 전략에 따라 위의 과정이 발생한다.

b. Deployment Descriptor안에 트랜잭션 속성을 텍스트로 지정하면 EJB 컨테이너가 자동으로 트랜잭션을 처리한다.

c. attribute -based programming

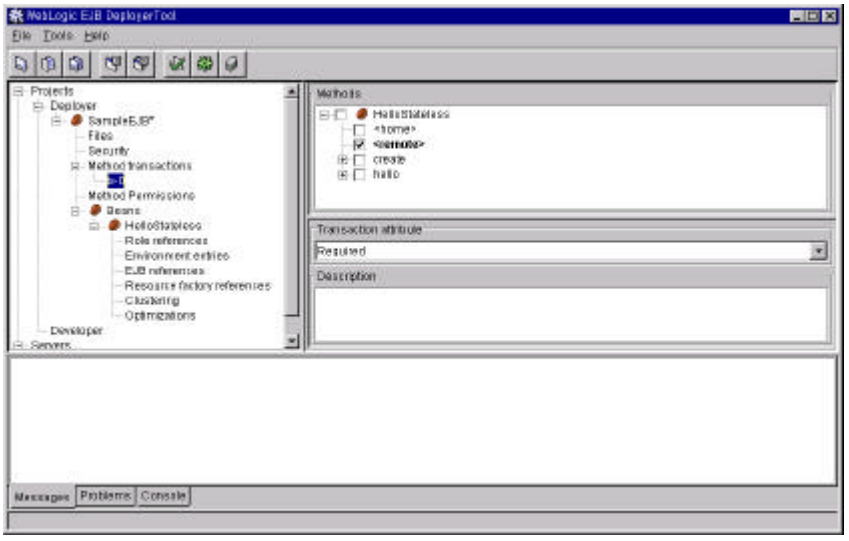
d. Declarative한 EJB 트랜잭션 실행 과정

EJB 빈 인스턴스가 아닌 자동생성된 EJB 객체가 트랜잭션을 지시한다.




<그림> declarative 트랜잭션

e. 웹로직 5.1에서 EJB빈을 Deploy할 때 트랜잭션 속성을 지정하는 과정



EJB 트랜잭션 속성

a. 누가 트랜잭션을 주도하는가?

EJB 클라이언트

EJB 컨테이너 (트랜잭션 자동화)

EJB 빈 컴포넌트

b. 6개의 트랜잭션 속성

Not Supported : EJB 컨테이너에서 트랜잭션을 지원하지 않음.

Required : EJB 메쏘드는 항상 트랜잭션 안에서 실행됨.

Supports : EJB 메쏘드가 트랜잭션안에 포함될 수도 있음.

RequiresNew : EJB 컨테이너는 항상 새로운 트랜잭션을 시도.

Mandatory : EJB 메쏘드 콜은 항상 트랜잭션안에 포함되며 클라이언트에 의해 관리됨.

Never : 결코 트랜잭션안에 포함되면 안됨.

c. 만약 EJB빈이 javax.ejb.SessionSynchronization 인터페이스를 구현했다면 빈의 트랜잭션 속성은 다음 중 1개로 지정해야 한다.

Required, RequiresNew, Mandatory

# Declarative 트랜잭션

## 주요 참고자

EJB빈 개발자가 아닌 EJB 빈을 이용하고 Deploy하는 사람이 트랜잭션 처리를 하고자 할 때 참조해야 함.

## EJB 트랜잭션 속성

EJB 빈 메쏘드마다 다음과 같은 트랜잭션 속성을 지정할 수 있다.

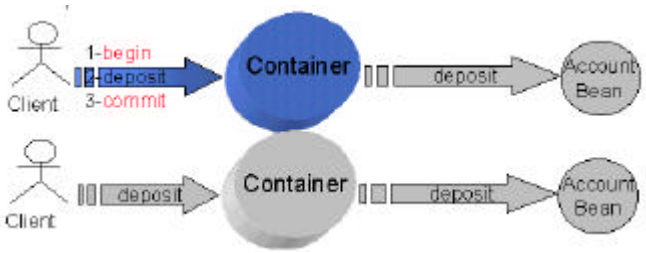
✍ Not Supported

EJB컨테이너는 명시적으로 트랜잭션을 지원하지 않는다.

EJB컨테이너는 자동으로 트랜잭션을 결코 실행하지 않는다. 또한 클라이언트안에서 트랜잭션을 시도해도 그것을 " 안 지원(Not Supported) " 한다.

만약 클라이언트 코드가 트랜잭션을 시도하면 EJB 메쏘드 호출이 끝날 때까지 클라이언트 트랜잭션은 멈춘 상태(suspend)가 된다.

만약 클라이언트 코드에 의해 호출된 EJB 빈이 다른 EJB빈을 호출했을 때 클라이언트의 트랜잭션 문맥(Transaction Context)은 전달되지 않는다.



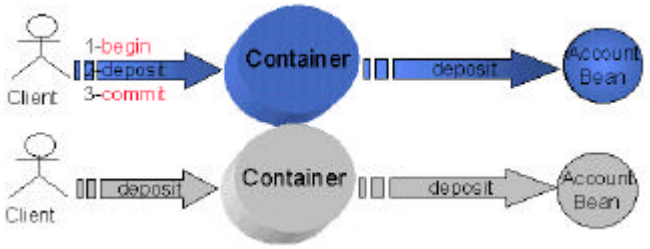

<그림> Not Supported EJB 트랜잭션

✍ Supports

EJB 컨테이너는 자동으로 트랜잭션을 결코 실행하지 않는다. 즉, 클라이언트가 트랜잭션을 시도하면 그것을 " 지원(Support) " 한다.

클라이언트 코드가 트랜잭션 하에서 EJB빈을 호출하면 " Required " 의 경우와 같이 동작한다.

클라이언트 코드가 트랜잭션이 아닌 상태에서 EJB빈을 호출하면 " Not Support " 와 경우와 같이 동작한다.




<그림> Supports EJB 트랜잭션

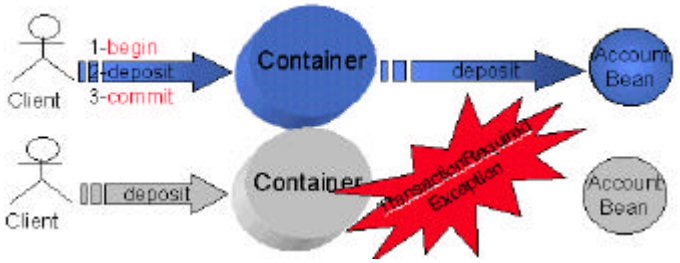
✍ Mandatory

클라이언트의 EJB 메쏘드 호출은 반드시 트랜잭션 하에서 이루어져야 한다. 트랜잭션은 클라이언트에 의해 관리되어져야 한다.

즉, 클라이언트는 트랜잭션 하에서 EJB 메쏘드 호출하도록 "강요(Mandatory)" 받는다. 그렇지 않으면 예외가 발생되기 때문이다.

EJB 컨테이너는 자동으로 트랜잭션을 결코 실행하지 않는다.

만약 클라이언트가 트랜잭션 하에서 EJB메쏘드 호출을 하지 않았을 경우에는 예외(TransactionRequiredException)가 발생된다.




<그림> Mandatory EJB 트랜잭션

✍ Required

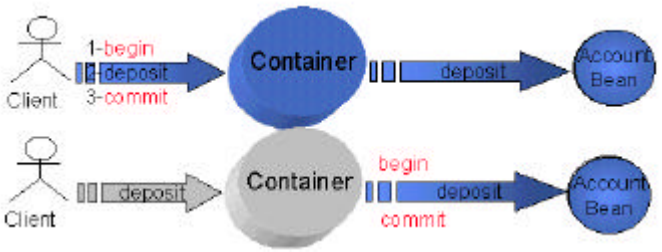
EJB빈 메쏘드는 항상 트랜잭션 상태에서 실행된다.

즉, EJB 빈 메쏘드는 컨테이너의 도움을 받아서 "필연적(Required)"으로 트랜잭션 하에서 실행된다.

클라이언트가 트랜잭션 하에서 동작하지 않으면 EJB 컨테이너는 자동으로 트랜잭션을 시작한다.

만약 호출된 EJB 빈 메쏘드가 다른 EJB메쏘드를 호출해도 해당 트랜잭션 문맥하에서 동작한다.

컨테이너는 EJB빈 메쏘드호출이 끝날 때 트랜잭션의 Commit을 시도한다. 또한 메쏘드의 결과값이 클라이언트에게 전달되기 전에 Commit 프로토콜을 수행한다.



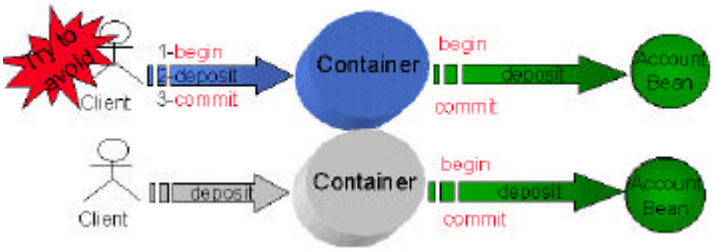

<그림> Required 트랜잭션

✍ RequiresNew

EJB 컨테이너는 EJB 빈 메쏘드를 호출하기 전에 항상 새로운 트랜잭션을 시작하며 메쏘드 호출이 끝났을 때 트랜잭션의 Commit을 시도한다.

즉, "필연적으로 항상 새로운(RequiresNew)" 트랙잭션이 EJB컨테이너에 의해서 시작된다.

만약 클라이언트 코드가 트랜잭션 하에서 EJB메쏘드를 호출하면 메쏘드 호출이 끝날 때까지 클라이언트 트랜잭션은 멈춘 상태(suspend)가 된다. 따라서 피하는 것이 좋다.




<그림> RequiresNew EJB 트랜잭션

✍ Never

트랜잭션이 "어떤 상황에서도 실행되지 않는다(Never)".

만약 클라이언트 코드가 트랜잭션 하에서 EJB메쏘드를 호출하면 java..rmi.RemoteException이 발생한다. 따라서 클라이어트는 트랜잭션 하에 있으면 안된다.

만약 클라이언트 코드가 트랙잭션 상태가 아니라면 "NotSupport" 경우와 같이 동작한다.

위의 6개 트랜잭션 속성을 요약한 테이블은 다음과 같다.

| 트랜잭션 속성 | 클라이언트의 트랜잭션 | EJB 빈 메쏘드와 연관된 트랜잭션 | 자원관리자와 연관된 트랜잭션 |
|---|---|---|---|
| NotSupported | 없음. | 없음. | 없음. |
| | T1 | 없음. | 없음. |
| Required | 없음. | T2 | T2 |
| | T1 | T1 | T1 |
| Supports | 없음. | 없음. | 없음. |
| | T1 | T1 | T1 |
| RequiresNew | 없음. | T2 | T2 |
| | T1 | T2 | T2 |
| Mandatory | 없음. | 에러 | N/A(Not Available) |
| | T1 | T1 | T1 |
| Never | 없음. | 없음. | 없음. |
| | T1 | 에러 | N/A |

T1 : 클라이어트코드 내에서 시작된 트랜잭션

T2 : EJB컨테이너에 의해 시작된 트랜잭션

## 트랜잭션 속성을 지정하는 구체적인 방법

✍✍ Deployment Descriptor에 지정

Deployment Descriptor에 메쏘드 단위로 6개의 EJB트랜잭션 속성을 지정할 수 있음.

```
<ejb-jar>
  …
  <assembly-descriptor>
    ...
    <container-transaction>
      <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
      </container-transaction>

      <container-transaction>
      <method>
      <ejb-name>EmployeeRecord</ejb-name>
      <method-name>updatePhoneNumber</method-name>
      </method>
      <trans-attribute>Mandatory</trans-attribute>
      </container-transaction>

      <container-transaction>
      <method>
      <ejb-name>AardvarkPayroll</ejb-name>
      <method-name>*</method-name>
      </method>
      <trans-attribute>RequiresNew</trans-attribute>
    </container-transaction>

  </assembly-descriptor>
</ejb-jar>
```

✍✍ Deployment 툴에서 GUI로 제공

EJB 서버마다 Deployment 툴에서 트랜잭션 속성을 비주얼하게 지정할 수 있음. 따라서 EJB 서버마다 지정하는 방식이 틀림.

웹로직의 경우는 "11.3"의 그림을 참고할 것

✍✍ **예제 소스코드**

**샘플 세션 빈 소스코드**

```
public class MySessionEJB implements SessionBean {
EJBContext ejbContext;
public void someMethod(...) {
  java.sql.Connection con1;
  java.sql.Connection con2;
  java.sql.Statement stmt1;
  java.sql.Statement stmt2;

  // obtain con1 and con2 connection objects
  con1 = ...;
  con2 = ...;
  stmt1 = con1.createStatement();
  stmt2 = con2.createStatement();

  /*
   Perform some updates on con1 and con2. The Container
  automatically enlists con1 and con2 with the container-
  managed transaction.
  */

  stmt1.executeQuery(...);
  stmt1.executeUpdate(...);
  stmt2.executeQuery(...);
  stmt2.executeUpdate(...);
  stmt1.executeUpdate(...);
  stmt2.executeUpdate(...);

  // release connections
  con1.close();
  con2.close();
  }
  ...
}
```

**주의할 점**

javax.ejb.SessionSynchronization 인터페이스

상태정보유지 세션빈은 본 인터페이스를 구현할 수도 있음.

javax.ejb.EJBContext.setRollbackOnly() 메쏘드

트랜잭션이 결코 Commit될 수 없도록 해주는 메쏘드. 어플리케이션 exception이 발생할 경우 자동으로 rollback이 되지 않기 때문에 강제로

rollback 해주는 기능이 필요.

javax.ejb.EJBContext.getRollbackOnly() 메쏘드

현재 트랜잭션이 setRollbackOnly()에 의해 rollback되도록 셋업되었는지 확인하는 메쏘드

# Programmatic 트랜잭션

## 주요 참고자

서버의 EJB 빈과 그 EJB빈을 호출하는 클라이언트 프로그램을 작성해야 하는 개발자. 트랜잭션 전문가가 효율적으로 프로그램을 구성할 때 적용.

## 서버의 EJB 빈 소스코드 내에서의 트랜잭션 코딩(bean-managed TX)

✍✍ EJB의 종류

세션빈 내에 트랜잭션 코드를 삽입하여 직접 관리한다. Entity 빈은 항상 EJB 컨테이너가 자동으로 트랜잭션을 관리해 주기 때문에 트랜잭션 코드를 삽입할 필요가 없다.

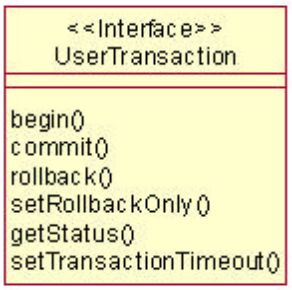
✍✍ JTA(Java Transaction API) 사용

javax.transaction.UserTransaction 사용

UserTransaction.begin()과 UserTransaction.commit()사이가 트랜잭션 범위가 됨.

트랜잭션 안에 java.sql.Connection의 commit(), rollback()메쏘드를 호출하면 안됨.

UserTransaction




트랜잭션 시작,Commit, rollback 수행함

트랜잭션의 commit을 강요함

트랜잭션의 상태정보를 얻을 수 있음

트랜잭션의 타임아웃(timeout)을 지정

JTS를 지원하는 환경에서, JNDI를 사용하여 javax.jts.UserTransaction 객체를 얻을 수 있음.

- context.lookup( “javax.jts.UserTransaction ”);

✍✍ **상태정보유지 세션 빈 VS 무상태 세션빈**

**가. 상태정보유지 세션빈은 EJB 빈 메쏘드가 끝날 때 반드시 트랜잭션을 Commit 할 필요는 없다.**

```
public class MySessionEJB implements SessionBean {
    EJBContext ejbContext;
    InitialContext initCtx;

    public void method1(...) {
            java.sql.Statement stmt;
            // obtain user transaction interface
            ut = ejbContext.getUserTransaction();
            // start a transaction
            ut.begin();
    }

    public void method2(...) {
            javax.sql.DataSource ds;
            java.sql.Connection con;
            java.sql.Statement stmt;
            // open connection
            ds = (javax.sql.DataSource)
            initCtx.lookup("java:comp/env/jdbcDatabase");
            con = ds.getConnection();
            // make some updates on con
            stmt = con.createStatement();
            stmt.executeUpdate(...);
            stmt.executeUpdate(...);
            // close the connection
            stmt.close();
            con.close();
    }

    public void method3(...) {
            // obtain user transaction interface
            ut = ejbContext.getUserTransaction();
            // commit the transaction
            ut.commit();
    }
```

```
    …
}
```

<소스> 여러 메쏘드에 걸친 트랜잭션

**나. 무상태 세션빈은 EJB 빈 메쏘드가 끝날 때 반드시 트랜잭션을 Commit해야 한다.**

```
public class MySessionEJB implements SessionBean {
    EJBContext ejbContext;
    public void someMethod(...) {
            javax.transaction.UserTransaction ut;
            javax.sql.DataSource ds1;
            javax.sql.DataSource ds2;
            java.sql.Connection con1;
            java.sql.Connection con2;
            java.sql.Statement stmt1;
            java.sql.Statement stmt2;
            InitialContext initCtx = new InitialContext();
            // obtain con1 object and set it up for transactions
            ds1 = (javax.sql.DataSource)
            initCtx.lookup("java:comp/env/jdbcDatabase1");
            con1 = ds1.getConnection();
            stmt1 = con1.createStatement();
            // obtain con2 object and set it up for transactions
            ds2 = (javax.sql.DataSource)
            initCtx.lookup("java:comp/env/jdbcDatabase2");
            con2 = ds2.getConnection();
            stmt2 = con2.createStatement();

            // Now do a transaction that involves con1 and con2.
            ut = ejbContext.getUserTransaction();

            // start the transaction
            ut.begin();
            stmt1.executeQuery(...);
            stmt1.executeUpdate(...);
            stmt2.executeQuery(...);
            stmt2.executeUpdate(...);
            stmt1.executeUpdate(...);
            stmt2.executeUpdate(...);
```

```
        // commit the transaction
        ut.commit();
        // release connections
        stmt1.close();
        stmt2.close();
        con1.close();
        con2.close();
}
...
```

<소스> 메쏘드가 끝날 때 Commit 시도

주의할 점

EJBContext의 getRollbackOnly()와 setRollbackOnly()를 사용해서는 안된다.

✍✍ 상태정보유지 세션빈(Stateful Session Bean)에서 SessionSynchronization을 사용하는 경우

가. 동기

상태정보유지 세션 빈에서 트랜잭션을 주체적으로 시도하고 관리하는 기능이 아니라 이미 수행되는 트랜잭션에서 begin, commit, rollback 등이 발생될 때 특정 기능이 수행되야 할 경우에 사용한다.

나. 구현

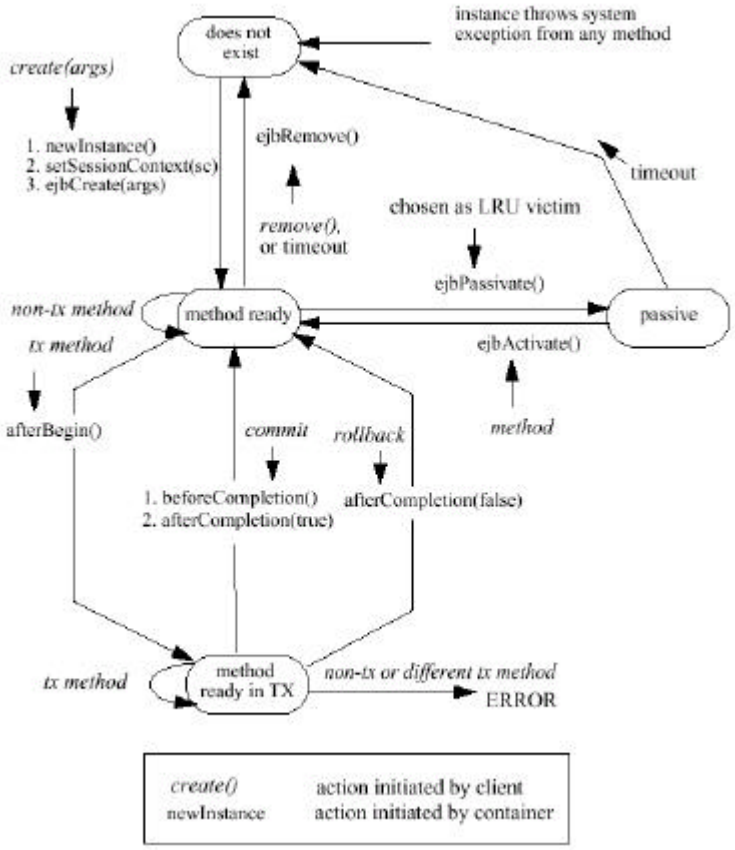
상태정보유지 세션빈 클래스에 SessionSynchronization 인터페이스를 구현한다.

afterBegin() : 트랜잭션이 시작한 직후에 컨테이너에 의해 호출됨.

beforeComletion() : 트랜잭션이 끝난 직전에 호출됨.

afterCompletion(boolean b) : 트랜잭션이 끝난 직후에 호출됨. 패러미터가 참이면 commit, 거짓이면 rollback이 발생한 것임.

다. 상태정보유지 세션빈 인스턴스의 생명주기




**<그림> 상태정보유지 세션빈의 트랜잭션 관련 라이프 사이클**

라. 사례연구

아래 소스에서 count()메쏘드는 Deployment Descriptor에 transaction 속성이 지정되어 있다고 가정함.

```
public class CountBean implements SessionBean, essionSynchronization {

   private SessionContext ctx;
   public int val;

   public void ejbCreate(int val) throws CreateException {
    this.val = val;
   }
   public int count() {
    return ++val;
   }
   public void afterCompletion(boolean b) {
    if (b == false) --val;
   }
   public void afterBegin() {}
   public void beforeCompletion() {}
   public void ejbRemove() {}
   public void ejbActivate() { }
   public void setSessionContext(SessionContext ctx) {}
}
```

<소스> CountBean.java

## 2. 클라이언트 소스코드 내에서의 트랜잭션 코딩

✍✍ JNDI 사용

JNDI를 사용하여 JTA 객체들(javax.transaction.*)을 얻은 후 메쏘드를 사용함.

✍✍ 관련 소스

웹로직에서의 JNDI 사용예

```
try {
    /*
    * 1: Set environment up. You must set the JNDI Initial
    * Context factory, the Provider URL, and any login
    * names or passwords necessary to access JNDI. See
```

```
        * your application server product's documentation for
        * details on their particular JNDI settings.
        */
        java.util.Properties env = ...

        /*
        * 2: Get the JNDI initial context
        */
        Context ctx = new InitialContext(env);

        /*
        * 3: Look up the JTA UserTransaction interface
        * via JNDI
        */
        userTran = (javax.transaction.UserTransaction)
        ctx.lookup("javax.transaction.UserTransaction");

        /*
        * 4: Execute the transaction
        */
        userTran.begin();

        // perform business operations

        userTran.commit();

        }
        catch (Exception e) {
        // deal with any exceptions, including ones
        // indicating an abort.
    }
```

<소스> 웹로직에서의 JNDI사용하는 클라이언트 코드

# Isolation과 Locking

## Isolation

✍✍ 의미

여러 사용자가 동시에 같은 데이터베이스 데이터를 사용해도 서로 독립적인 데이터 처리를 할 수 있는 능력

✍✍ 일반적인 레벨

Dirty Read

A 트랜잭션이 B 트랜잭션이 변동한 데이터를 Commit하기 전에 읽는다. 만약 B 트랜잭션이 rollback되면 A 트랜잭션은 무효한 경우가 된다.

Repeatable Read

한 트랜잭션이 수행중에 읽은 데이터는 트랜잭션 중이라면 이후에 읽어도 같은 값을 가진다.

Phantom Read

데이터베이스에 데이터를 삽입하기 전에 시작한 트랜잭션은 데이터베이스에 새로 첨가된 레코드들을 알 수 없다.

## EJB와 Isolation

✍✍ EJB에 명시되어 있지 않고 Resource Manager에 따라 달라진다. 즉 Resource Manager가 데이터베이스인 경우 해당 데이터베이스가 지원하는 Isolation과 locking에 따라 실제 정확한 Isolatin 수행 행위가 결정된다.

✍✍ Deployment Descriptor에 메쏘드 단위로 6개의 EJB트랜잭션 속성을 지정할 수 있음.

✍✍ JDBC의 트랜잭션 isolation을 사용한 EJB Isolation

| Isolation레벨 | 설명 |
|---|---|
| TRANSACTION_READ_UNCOMMITEED | Commit되지 않은 데이터들을 읽을 수 있다. |
| TRANSACTION_READ_COMMITTED | Commit되지 않은 데이터들을 읽을 수 없다. |
| TRANSACTION_REPEATABLE_READ | 어떤 트랜잭션도 다른 트랜잭션에 의해서 읽혀지고 있는 데이터를 변경할 수 없다. |
| TRANSACTION_SERIALZABLE | 어떤 트랜잭션도 다른 트랜잭션이 사용하고 있는 데이터를 읽을 수도 변경할 수도 없다. |

✍✍ 요약

| Isolation레벨 | Dirty Reads? | UnReapeatable Reads? | Phantom Reads? |
|---|---|---|---|
| TRANSACTION_READ_UNCOMMITEED | 예 | 예 | 예 |
| TRANSACTION_READ_COMMITTED | 아니오 | 예 | 예 |
| TRANSACTION_REPEATABLE_READ | 아니오 | 아니오 | 예 |
| TRANSACTION_SERIALZABLE | 아니오 | 아니오 | 아니오 |

# 예외 처리

## EJB 컨테이너에 의해 자동으로 시작된 트랜잭션의 예외처리

- 기본적으로 트랜잭션의 범위밖으로 던져지는 어떤 예외(어플리케이션 예외, RMI예외 등)은 컨테이너가 rollback 처리한다.
- 반면, 트랜잭션 범위안에 던져지는 예외는 프로그래머가 rollback을 위한 예외 처리를 해줘야 한다.

| Method condition | Method exception | Container's action | Client's view |
|---|---|---|---|
| Bean method runs in the context of the caller's transaction [Note A]. This case may happen with `Required`, `Mandatory`, and `Supports` attributes. | AppException | Re-throw AppException | Receives AppException.<br>Can attempt to continue computation in the transaction, and eventually commit the transaction (the commit would fail if the instance called `setRollbackOnly()`). |
| | all other exceptions and errors | Log the exception or error [Note B].<br>Mark the transaction for rollback.<br>Discard instance [Note C].<br>Throw `TransactionRolledBackException` to the client. | Receives `TransactionRolledBackException`.<br>Continuing transaction is fruitless. |
| Bean method runs in the context of a transaction that the Container started immediately before dispatching the business method. This case may happen with `Required` and `RequiresNew` attributes. | AppException | If the instance called setRollbackOnly(), then rollback the transaction, and re-throw AppException.<br>Otherwise, attempt to commit the transaction, and then re-throw AppException. | Receives AppException.<br>If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work. |
| | all other exceptions | Log the exception or error.<br>Rollback the container-started transaction.<br>Discard instance.<br>Throw `RemoteException`. | Receives `RemoteException`.<br>If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work. |

| Method condition | Method exception | Container's action | Client's view |
|---|---|---|---|
| Bean method runs with an unspecified transaction context. This case may happen with the `NotSupported`, `Never`, and `Supports` attributes. | AppException | Re-throw AppException. | Receives AppException.<br>If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work. |
| | all other exceptions | Log the exception or error.<br>Discard instance.<br>Throw `RemoteException`. | Receives `RemoteException`.<br>If the client executes in a transaction, the client's transaction is not marked for rollback, and client can continue its work. |

### EJB빈 코드 내에서 시작한 트랜잭션의 예외처리

| Bean method condition | Bean method exception | Container action | Client receives |
|---|---|---|---|
| Bean is stateful or stateless Session. | AppException | Re-throw AppException | Receives AppException. |
| | all other exceptions | Log the exception or error.<br>Mark for rollback a transaction that has been started, but not yet completed, by the instance.<br>Discard instance.<br>Throw `RemoteException`. | Receives `RemoteException`. |

### 클라이언트 코드 내에서 시작된 트랜잭션의 예외처리

예외가 발생했을 때 클라이언트는 rollback을 위한 예외처리 코드를 작성해야 한다.

# 사례 연구

## EJB 컨테이너에 의해 자동으로 시작 및 관리되는 트랜잭션 예제

### 내용

✍✍ 은행업무 처리 예제

Bank.java, BankClient.java, BankEJB.java, BankHome.java,
createTable.sql, InsufficientBalanceException.java

### 소스

✍✍ bank.java : 리모트 인터페이스

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Bank extends EJBObject {

   public void transferToSaving(double amount)
      throws RemoteException;

   public double getCheckingBalance()
      throws RemoteException;

   public double getSavingBalance()
      throws RemoteException;
}
```

<소스> Bank.java

✍✍ BankEJB.java : EJB 빈 클래스

```
import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class BankEJB implements SessionBean, SessionSynchronization {

  private String customerId;
```

```
private double checkingBalance;
private double savingBalance;
private SessionContext context;
private Connection con;
private String dbName = "java:comp/env/jdbc/BankDB";

public void transferToSaving(double amount) throws
  InsufficientBalanceException  {

  checkingBalance -= amount;
  savingBalance += amount;

  try {
    updateChecking(checkingBalance);
    if (checkingBalance < 0.00) {
      context.setRollbackOnly();
      throw new InsufficientBalanceException();
    }
    updateSaving(savingBalance);
  } catch (SQLException ex) {
     throw new EJBException
       ("Transaction failed due to SQLException: "
       + ex.getMessage());
  }
}

public double getCheckingBalance() {

  return checkingBalance;
}

public double getSavingBalance() {

  return savingBalance;
}

public void ejbCreate(String id) throws CreateException {

  customerId = id;
```

```
        try {
           makeConnection();
           checkingBalance = selectChecking();
           savingBalance = selectSaving();
        } catch (Exception ex) {
            throw new CreateException(ex.getMessage());
        }

    }


    public void ejbRemove() {

        try {
            con.close();
        } catch (SQLException ex) {
            throw new EJBException("ejbRemove SQLException: " +
ex.getMessage());
        }
    }

    public void ejbActivate() {

        try {
           makeConnection();
        } catch (Exception ex) {
            throw new EJBException("ejbActivate Exception: " + ex.getMessage());
        }
    }

    public void ejbPassivate() {

        try {
          con.close();
        } catch (SQLException ex) {
            throw new EJBException("ejbPassivate Exception: " + ex.getMessage());
        }
    }
```

```
public void setSessionContext(SessionContext context) {
   this.context = context;
}

public void afterBegin() {

   System.out.println("afterBegin()");
   try {
      checkingBalance = selectChecking();
      savingBalance = selectSaving();
   } catch (SQLException ex) {
       throw new EJBException("afterBegin Exception: " + ex.getMessage());
   }

}

public void beforeCompletion() {

   System.out.println("beforeCompletion()");
}

public void afterCompletion(boolean committed) {

   System.out.println("afterCompletion: " + committed);
   if (committed == false) {
      try {
         checkingBalance = selectChecking();
         savingBalance = selectSaving();
      } catch (SQLException ex) {
          throw new EJBException("afterCompletion SQLException: " +
             ex.getMessage());
      }
   }
}

public BankEJB() {}
```

```
/****************** Database Routines **********************/

  private void updateChecking(double amount) throws SQLException {

    String updateStatement =
        "update checking set balance =  ? " +
        "where id = ?";

    PreparedStatement prepStmt =
        con.prepareStatement(updateStatement);

    prepStmt.setDouble(1, amount);
    prepStmt.setString(2, customerId);
    prepStmt.executeUpdate();
    prepStmt.close();
  }

  private void updateSaving(double amount) throws SQLException {

    String updateStatement =
        "update saving set balance =  ? " +
        "where id = ?";
    PreparedStatement prepStmt =
        con.prepareStatement(updateStatement);

    prepStmt.setDouble(1, amount);
    prepStmt.setString(2, customerId);
    prepStmt.executeUpdate();
    prepStmt .close();
  }

  private double selectChecking() throws SQLException {

    String selectStatement =
        "select balance " +
        "from checking where id = ? ";
    PreparedStatement prepStmt =
        con.prepareStatement(selectStatement);
```

```
    prepStmt.setString(1, customerId);

    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
      double result = rs.getDouble(1);
      prepStmt.close();
      return result;
    }
    else {
      prepStmt.close();
      throw new EJBException
        ("Row for id " + customerId + " not found.");
    }
  }

  private double selectSaving() throws SQLException {

    String selectStatement =
        "select balance " +
        "from saving where id = ? ";
    PreparedStatement prepStmt =
        con.prepareStatement(selectStatement);

    prepStmt.setString(1, customerId);

    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
      double result = rs.getDouble(1);
      prepStmt.close();
      return result;
    }
    else {
      prepStmt.close();
      throw new EJBException
        ("Row for id " + customerId + " not found.");
    }
```

```
  }

  private void makeConnection()
    throws NamingException, SQLException {

    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup(dbName);
    con =  ds.getConnection();
  }

} // BankEJB
```

<소스> BankEJB.java

다. BankHome.java

```
import java.rmi.RemoteException;
import javax.ejb.*;

public interface BankHome extends EJBHome {

   public Bank create(String id)
      throws RemoteException, CreateException;
}
```

<소스> BankHome.java

✍ InsufficientBalanceException.java

```
public class InsufficientBalanceException extends Exception {

  public InsufficientBalanceException() { }

  public InsufficientBalanceException(String msg) {
    super(msg);
  }
}
```

<소스> InsufficientBalanceException.java

✍ createTable.sql

```
drop table checking;

create table checking
(id varchar(3) constraint pk_checking primary key,  balance decimal(10,2));

insert into checking  values ('123', 100.00);

drop table saving;

create table saving  (id varchar(3) constraint pk_saving primary key,  balance
decimal(10,2));

insert into saving  values ('123', 500.00);

exit;
```

✍ BankClient.java : **클라이언트 코드**

```
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class BankClient {

  public static void main(String[] args) {

      try {
         Context initial = new InitialContext();
         Object objref = initial.lookup("MyBank");
         BankHome home =
               (BankHome)PortableRemoteObject.narrow(objref,
BankHome.class);
         Bank duke = home.create("123");
         duke.transferToSaving(40.00);
         System.out.println("checking: " + duke.getCheckingBalance());
         System.out.println("saving: " + duke.getSavingBalance());
         duke.remove();
```

```
        } catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
    }
}
```

<소스> BankClient.java

## EJB 빈에서 JDBC를 사용하여 시작 및 관리되는 트랜잭션 예제

### 내용

✍✍ **창고관리 시스템**

✍✍ Warehouse.java, WarehouseClient.java, WarehouseEJB.java, WarehouseHome.java, createTable.sql

### 소스

✍✍ Warehouse.java : **리모트 인터페이스**

```
import java.rmi.RemoteException;
import javax.ejb.*;

public interface WarehouseHome extends EJBHome {

    public Warehouse create()
        throws RemoteException, CreateException;
}
```

<소스> Warehouse.java

✍✍ WarehouseEJB.java : EJB **빈 클래스**

```
import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class WarehouseEJB implements SessionBean {

    private SessionContext context;
    private Connection con;
    private String dbName = "java:comp/env/jdbc/WarehouseDB";
```

```
public void ship (String productId, String orderId, int quantity) {

  try {
    con.setAutoCommit(false);
    updateOrderItem(productId, orderId);
    updateInventory(productId, quantity);
    con.commit();
  } catch (Exception ex) {
     try {
       con.rollback();
       throw new EJBException("Transaction failed: " + ex.getMessage());
     } catch (SQLException sqx) {
        throw new EJBException("Rollback failed: " + sqx.getMessage());
     }
  }
}

public String getStatus(String productId, String orderId) {

  try {
    return selectStatus(productId, orderId);
  } catch (SQLException ex) {
     throw new EJBException
       ("Unable to fetch status due to SQLException: "
       + ex.getMessage());
  }
}

public void ejbCreate() throws CreateException {

  try {
    makeConnection();
  } catch (Exception ex) {
     throw new CreateException(ex.getMessage());
  }

}
```

```
  public void ejbRemove()  {

    try {
      con.close();
    } catch (SQLException ex) {
       throw new EJBException(ex.getMessage());
    }
  }

  public void ejbActivate()  {

    try {
      makeConnection();
    } catch (Exception ex) {
       throw new EJBException(ex.getMessage());
    }
  }

  public void ejbPassivate()  {

    try {
      con.close();
    } catch (SQLException ex) {
       throw new EJBException(ex.getMessage());
    }
  }

  public void setSessionContext(SessionContext context) {

    this.context = context;
  }

  public WarehouseEJB() {}

/********************** Database Routines ************************/

  private String selectStatus(String productId, String orderId)
    throws SQLException {
```

```
    String result;

    String selectStatement =
        "select status " +
        "from order_item where product_id = ? " +
        "and order_id = ?";
    PreparedStatement prepStmt =
        con.prepareStatement(selectStatement);

    prepStmt.setString(1, productId);
    prepStmt.setString(2, orderId);

    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
      result = rs.getString(1);
    }
    else {
      result = "No rows found.";
    }

    prepStmt.close();
    return result;
  }

  private void updateOrderItem(String productId, String orderId)
    throws SQLException {

    String updateStatement =
        "update order_item set status =  'shipped' " +
        "where product_id = ? " +
        "and order_id = ?";

    PreparedStatement prepStmt =
        con.prepareStatement(updateStatement);

    prepStmt.setString(1, productId);
    prepStmt.setString(2, orderId);
    prepStmt.executeUpdate();
```

```
        prepStmt.close();
    }

    private void updateInventory(String productId, int quantity)
        throws SQLException {

        String updateStatement =
            "update inventory " +
            "set quantity = quantity - ? " +
            "where product_id = ?";

        PreparedStatement prepStmt =
            con.prepareStatement(updateStatement);

        prepStmt.setInt(1, quantity);
        prepStmt.setString(2, productId);
        prepStmt.executeUpdate();
        prepStmt.close();
    }

    private void makeConnection()
        throws NamingException, SQLException {

        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(dbName);
        con =  ds.getConnection();
    }

} // WarehouseEJB;
```

<소스> WarehouseEJB.java

✍✍ WarehouseHome.java

```
import java.rmi.RemoteException;
import javax.ejb.*;

public interface WarehouseHome extends EJBHome {

    public Warehouse create()
        throws RemoteException, CreateException;
}
```

<소스> WareHouseHome.java

✍✍ createTable.sql

```
drop table inventory;

create table inventory  (product_id varchar(3),  quantity decimal(10));

insert into inventory  values ('123', 100);

drop table order_item;

create table order_item
 (order_id varchar(3),  product_id varchar(3),  status varchar(8));

insert into order_item  values ('456', '123', 'open');

exit;
```

✍✍ WareHouseClient.java : **클라이언트 코드**

```
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class WarehouseClient {

   public static void main(String[] args) {
```

```
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup("MyWarehouse");

            WarehouseHome home =
                (WarehouseHome)PortableRemoteObject.narrow(objref,
                                    WarehouseHome.class);

            Warehouse duke = home.create();
            duke.ship("123", "456", 8);
            System.out.println("status = " + duke.getStatus("123", "456"));
            duke.remove();

        } catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }

    }
}
```

<소스> WarehouseClient.java

## EJB 빈에서 JTA를 사용하여 시작 및 관리되는 트랜잭션 예제

### 내용

✍✍ ATM 시스템

✍✍ Teller.java, TellerClient.java, TellerEJB.java, TellerHome.java, createTable.sql

### 소스

✍✍ Teller.java : 리모트 인터페이스

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
```

```
public interface Teller extends EJBObject {

  public void withdrawCash(double amount)
     throws RemoteException;

  public double getCheckingBalance()
     throws RemoteException;
}
```

<소스> Teller.java

✍✍ TellerEJB.java : EJB 빈 클래스

```
import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.transaction.*;

public class TellerEJB implements SessionBean {

  private String customerId;
  private double machineBalance;
  private SessionContext context;
  private Connection con;
  private String dbName = "java:comp/env/jdbc/TellerDB";

  public void withdrawCash(double amount) {

    UserTransaction ut = context.getUserTransaction();

    try {
      ut.begin();
      updateChecking(amount);
      machineBalance -= amount;
      insertMachine(machineBalance);
      ut.commit();
    } catch (Exception ex) {
       try {
          ut.rollback();
       } catch (SystemException syex) {
           throw new EJBException
             ("Rollback failed: " + syex.getMessage());
       }
       throw new EJBException
          ("Transaction failed: " + ex.getMessage());
  }
 }
```

```
public double getCheckingBalance() {

  try {
    return selectChecking();
  } catch (SQLException ex) {
     throw new EJBException
       ("Unable to get balance: "
       + ex.getMessage());
  }
}

public void ejbCreate(String id) throws CreateException {

  customerId = id;

  try {
    makeConnection();
    machineBalance = selectMachine();
  } catch (Exception ex) {
     throw new CreateException(ex.getMessage());
  }

}

public void ejbRemove()  {

  try {
    con.close();
  } catch (SQLException ex) {
     throw new EJBException(ex.getMessage());
  }
}

public void ejbActivate()  {

  try {
    makeConnection();
  } catch (Exception ex) {
     throw new EJBException(ex.getMessage());
```

```
    }
  }

  public void ejbPassivate()  {

    try {
      con.close();
    } catch (SQLException ex) {
       throw new EJBException(ex.getMessage());
    }
  }

  public void setSessionContext(SessionContext context) {
    this.context = context;
  }

  public TellerEJB() {}

/************************ Database Routines **********************/

  private void makeConnection()
    throws NamingException, SQLException {

    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup(dbName);
    con =  ds.getConnection();
  }

  private void updateChecking(double amount)
    throws SQLException {

    String updateStatement =
        "update checking set balance =  balance - ? " +
        "where id = ?";

    PreparedStatement prepStmt =
        con.prepareStatement(updateStatement);

    prepStmt.setDouble(1, amount);
```

```
    prepStmt.setString(2, customerId);
    prepStmt.executeUpdate();
    prepStmt.close();
  }

  private void insertMachine(double amount)
    throws SQLException {

    String insertStatement =
        "insert into cash_in_machine values " +
        "( ? , current_date )";

    PreparedStatement prepStmt =
        con.prepareStatement(insertStatement);

    prepStmt.setDouble(1, amount);
    prepStmt.executeUpdate();
    prepStmt.close();
  }

  private double selectMachine() throws SQLException  {

    String selectStatement =
        "select amount " +
        "from cash_in_machine " +
        "where time_stamp = " +
        "(select max(time_stamp) from cash_in_machine)";
    PreparedStatement prepStmt =
        con.prepareStatement(selectStatement);

    ResultSet rs = prepStmt.executeQuery();

    if (rs.next()) {
      double result = rs.getDouble(1);
      prepStmt.close();
      return result;
    }
    else {
      prepStmt.close();
```

```
            throw new EJBException
              ("Row for id " + customerId + " not found.");
          }
        }

        private double selectChecking() throws SQLException  {

          String selectStatement =
              "select balance " +
              "from checking " +
              "where id = ?";
          PreparedStatement prepStmt =
              con.prepareStatement(selectStatement);

          prepStmt.setString(1, customerId);
          ResultSet rs = prepStmt.executeQuery();

          if (rs.next()) {
            double result = rs.getDouble(1);
            prepStmt.close();
            return result;
          }
          else {
            prepStmt.close();
            throw new EJBException
              ("Row for id " + customerId + " not found.");
          }
        }

      } // TellerEJB
```

<소스> TellerEJB.java

✍ TellerHome.java

```
import java.rmi.RemoteException;
import javax.ejb.*;

public interface TellerHome extends EJBHome {
```

```
    public Teller create(String id)
        throws RemoteException, CreateException;
}
```

<소스> TellerHome.java

✍ createTable.sql

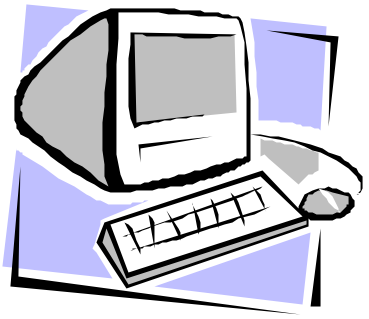
```
drop table checking;

create table checking
(id varchar(3) constraint pk_checking primary key,  balance decimal(10,2));

insert into checking  values ('123', 500.00);

drop table cash_in_machine;

create table cash_in_machine  (amount decimal(10,2),  time_stamp date);

insert into cash_in_machine  values (10000.00, current_date);

exit;
```

✍ TellerClient.java : **클라이언트 코드**

```
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class TellerClient {

  public static void main(String[] args) {

      try {
         Context initial = new InitialContext();
         Object objref = initial.lookup("MyTeller");

         TellerHome home =
            (TellerHome)PortableRemoteObject.narrow(objref,
```

```
                                        TellerHome.class);

            Teller duke = home.create("123");
            System.out.println("checking = " + duke.getCheckingBalance());
            duke.withdrawCash(60.00);
            System.out.println("checking = " + duke.getCheckingBalance());
            duke.remove();

        } catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
    }
}
```

<소스> TellerClient.java



# Chapter 13

# Descriptor

## 배치 디스크립터 (Deployment Descriptor)

배치 디스크립터는 ejb-jar 파일의 생산자와 소비자사이의 약속의 한 부분이다. 이 계약은 빈 제공자로부터 애플리케이션 조립자로의, 그리고 애플리케이션 조립자로부터 배치자로의 엔터프라이즈 빈 전달을 담당한다.

빈 제공자에의해 생성된 ejb-jar 파일은 하나 또는 그이상의 엔터프라이즈 빈을 포함하며 일반적으로 애플리케이션 조립 명령은 포함하지 않는다. 애플리케이션 조립자에 의해 생성된 ejb-jar 파일은 하나 또는 그 이상의 엔터프라이즈 빈을 포함하고 더불어 엔터프라이즈 빈들이 어떻게 하나의 애플리케이션 배치 유닛(unit)으로 결합되는지를 묘사한 애플리케이션 조립 정보를 포함한다. J2EE 스펙은 엔터프라이즈 빈과 여러 개의 ejb-jar 파일에 포함된 다른 애플리케이션 컴포넌트가 어떻게 애플리케이션으로 결합되는지 정의한다.

배치 디스크립터의 역할은 ejb-jar 파일의 사용자를 위해 제공되는 선언적인 정보(즉, 엔터프라이즈 빈의 코드에는 직접적으로 포함되지 않은 정보)를 획득하는 것이다.

배치 디스크립터에는 두 개의 종류의 기본적인 정보가 있다.

### 엔터프라이즈 빈의 구조적인 정보

✍ 구조적인 정보는 엔터프라이즈 빈의 구조를 설명하고 엔터프라이즈 빈의 외부 의존도를 선언한다. 배치 디스크립터에 있어서 구조적인 정보의 제공은 ejb-jar 파일 생산자를 위해서 필수적이다. 일반적으로 구조적인 정보는 바뀔 수 없다. 그렇게 하는 것이 엔터프라이즈 빈의 기능을 망가뜨릴 수 있기 때문이다.

### 애플리케이션 조립 정보

✍ 애플리케이션 조립 정보는 ejb-jar 파일에서 엔터프라이즈 빈이 어떻게 더 큰 애플리케이션 배치 유닛으로 구성 되는지 보여준다. 배치 디스크립트에서 조립 정보를 제공하는 것은 ejb-jar 파일 생산자에게는 선택 사항이다. 그렇게 하는 것이 조합된 애플리케이션의 동작을 바꿀지도 모르지만 조립 레벨 정보는 엔터프라이즈 빈의 기능을 바꾸지 않고 변경 가능하다.

## 각 역할 분담자의 책임

### 빈 제공자의 책임

빈 제공자는 배치 디스크립트에 있어서 각 엔터프라이즈 빈에 대한 구조적인 정보를 제공할 책임이 있다.

빈 제공자는 ejb-jar 파일에서 모든 엔터프라이즈 빈을 나열하기 위해 enterprise-beans 엘리먼트를 사용해야한다.

빈 제공자는 각 엔터프라이즈 빈마다 다음과 같은 정보를 제공해야 한다.

✍ 엔터프라이즈 빈의 이름

빈 제공자는 논리적인 이름을 ejb-jar 파일에 있는 각각의 엔터프라이즈 빈에 할당해야 한다. 이 이름과 배치자가 엔터프라이즈 빈에 할당한 JNDI API 이름 사이에는 어떤 구조화된 관계도 없다. 빈 제공자는 ejb-name 엘리먼트에서 엔터프라이즈 빈의 이름을 기술한다.

✍ 엔터프라이즈 빈의 클래스

빈 제공자는 엔터프라이즈 빈의 비즈니스 메소드를 구현하는 자바 클래스의 이름을 기술해야 한다. 빈 제공자는 ejb-class 엘리먼트에 엔터프라이즈 빈의 이름을 기술한다.

✍ 엔터프라이즈 빈의 홈 인터페이스

빈 제공자는 home 엘리먼트에 엔터프라이즈 빈 홈 인터페이스의 이름을 기술해야 한다.

✍ 엔터프라이즈 빈의 리모트 인터페이스

빈 제공자는 remote 엘리먼트에 엔터프라이즈 빈 리모트 인터페이스의 이름을 기술해야 한다.

✍ 엔터프라이즈 빈의 종류

엔터프라이즈 빈의 종류로는 세션 빈과 엔티티 빈이 있다. 빈 제공자는 엔터프라이즈 빈의 구조 정보를 선언하기 위해서 session 이나 entity 엘리먼트 중 알맞은 엘리먼트를 사용한다.

✍ 재진입 표시 (Re-entrancy indication)

빈 제공자는 엔티티 빈에 대한 재진입이 가능한지 불가능한지를 기술해야 한다.

세션 빈은 재진입이 불가능하다.

✍✍ 세션 빈의 상태 관리 종류

만약 엔터프라이즈 빈이 세션 빈이라면 빈 제공자는 세션 빈이 상태유지 세션 빈인지 무상태 세션 빈인지를 session-type 엘리먼트를 이용해 선언해야 한다.

✍✍ 엔티티 빈의 퍼시스턴스 관리

만약 엔터프라이즈 빈이 엔티티 빈이라면 빈 제공자는 엔티티 빈의 퍼시스턴스 관리가 엔터프라이즈 빈에 의해 이루어 지는지 컨테이너에 의해 이루어 지는지를 persistence-type 엘리먼트를 사용해 선언해야 한다.

✍✍ 엔터프라이즈 빈의 프라이머리 키 클래스

엔터프라이즈 빈이 엔티티 빈이라면 빈 제공자는 엔티티 빈의 프라이머리 키 클래스의 이름을 prim-key-class 엘리먼트에 기술한다. 빈 제공자는 엔터프라이즈 빈이 빈 관리 엔티티 빈일 경우 반드시 프라이머리 키 클래스를 기술해야 하며, 컨테이너 관리 엔티티 빈일 경우 프라이머리 키를 기술하지 않을 수도 있다.

✍✍ 컨테이너 관리 필드

엔터프라이즈 빈이 컨테이너 관리 퍼시스턴스를 가질 경우 빈 제공자는 cmp-fields 엘리먼트를 사용하여 컨테이너 관리 필드를 기술해야 한다.

✍✍ 환경 엔트리 (Environment entries)

빈 제공자는 모든 엔터프라이즈 빈의 엔트리를 선언해야 한다.

✍✍ 리소스 매니저 컨넥션 팩토리 레퍼런스(Resource manager connection factory references)

빈 제공자는 모든 엔터프라이즈 빈의 리소스 매니저 컨넥션 팩토리 레퍼런스를 선언해야 한다.

✍✍ EJB 레퍼런스

빈 제공자는 모든 엔터프라이즈 빈의 다른 엔터프라이즈 빈의 홈에 대한 레퍼런스를 선언해야 한다.

✍✍ 보안 역할 레퍼런스 (Security role references)

빈 제공자는 모든 엔터프라이즈 빈의 보안 역할에 대한 레퍼런스를 제공해야 한다.

빈 제공자에 의해서 생성된 배치 디스크립터는 XML 양식에서 잘 만들어져야 하며 DTD의 관점에서 유효해야 한다. 배치 디스크립터는 다음과 같은 문장을 이용해서 DTD를 참고해야 한다.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

## 애플리케이션 조립자의 책임

애플리케이션 조립자는 엔터프라이즈 빈들을 하나의 배치 유닛으로 조립한다. 애플리케이션 조립자의 입력은 하나 이상의 빈 제공자에 의해 제공된 하나 또는 그

이상의 ejb-jar 파일들이다. 애플리케이션 조립자는 여러 개의 입력 ejb-jar 파일들을 하나의 출력 ejb-jar 파일로 묶을 수 있으며, 하나의 입력 ejb-jar 파일을 여러 개의 출력 ejb-jar 파일들로 나눌 수 있다. 각 출력 ejb-jar 파일은 배치자를 위한 배치 유닛이거나 또 다른 애플리케이션 조립자를 위한 부분적으로 조립된 애플리케이션이다.
빈 제공자와 애플리케이션 조립자는 같은 사람이거나 같은 조직에 속할 수 있다. 이러한 경우 그 사람이나 조직은 양쪽에서 기술된 책임들을 모두 수행해야 한다.
애플리케이션 조립자는 빈 제공자에 의해 기술된 다음의 정보들을 수정할 수도 있다.

✍✍ **엔터프라이즈 빈의 이름**

애플리케이션 조립자는 ejb-name 엘리먼트에 정의된 엔터프라이즈 빈의 이름을 변경할 수 있다.

✍✍ **환경 엔트리의 값들** (Values of environment entries)

애플리케이션 조립자는 이미 존재하는 환경 프로퍼티의 값을 바꿀 수 있으며 또한 새로운 값을 정의할 수도 있다.

✍✍ **디스크립션 필드** (Description fields)

애플리케이션 조립자는 이미 존재하는 description 엘리먼트를 변경하거나 새로 생성할 수 있다.

애플리케이션 조립자는 보통 입력 ejb-jar 파일에서 제공되는 정보들을 수정하면 안된다.
그리고 애플리케이션 조립자는 다음의 애플리케이션 조립 정보를 기술할 수 있다. (필수사항은 아니다.)

✍✍ **엔터프라이즈 빈 레퍼런스의 바인딩** (Binding of enterprise bean refrences)

애플리케이션 조립자는 엔터프라이즈 빈 레퍼런스를 ejb-jar 파일 내의 다른 엔터프라이즈 빈에 연결할 수 있다. 애플리케이션 조립자는 빈을 참조하기 위해서 ejb-link 엘리먼트를 추가하여 연결을 생성한다.

✍✍ **보안 역할** (Security roles)

애플리케이션 조립자는 하나 또는 그 이상의 보안 역할을 정의할 수도 있다. 보안 역할은 엔터프라이즈 빈의 클라이언트를 위해 추천(recommended) 보안 역할을 정의한다. 애플리케이션 조립자는 보안 역할을 security -role 엘리먼트를 사용하여 정의한다.

✍✍ **메소드 권한** (Method permissions)

애플리케이션 조립자는 메소드 권한을 정의할 수도 있다. 메소드 권한은 보안 역할과 엔터프라이즈 빈의 리모트 인터페이스 및 홈 인터페이스의 메소드와의 이원적인 관계이다. 애플리케이션 조립자는 method-permission 엘리먼트를 이용하여 메소드 권한을 정의한다.

✍✍ **보안 역할 레퍼런스의 연결** (Linking of security role references)

애플리케이션 조립자가 배치 디스크립터에서 보안 역할을 정의한다면 애플리케

이션 조립자는 빈 제공자에 의해 선언된 보안 역할 레퍼런스들을 보안 역할들에 연결해야 한다. 애플리케이션 조립자는 이러한 연결을 role-link 엘리먼트를 사용하여 정의한다.

✍✍ 트랜잭션 속성 (Transaction attributes)

애플리케이션 조립자는 컨테이너 관리 트랜잭션 분리가 필요한 엔터프라이즈 빈의 리모트 인터페이스 및 홈 인터페이스의 메소드에 대한 트랜잭션 속성의 값을 정의할 수도 있다. 빈 제공자에 의해 트랜잭션 종류가 Container로 선언된 모든 엔티티 빈과 세션 빈은 컨테이너 관리 트랜잭션 분리(demarcation)이 필요하다. 애플리케이션 조립자는 container-transaction 엘리먼트를 사용하여 트랜잭션 속성을 선언한다.

입력 ejb-jar 파일이 애플리케이션 조립 정보를 포함하고 있다면 애플리케이션 조립자는 입력 ejb-jar 파일의 애플리케이션 조립 정보를 변경할 수 있다. (이와 같은 경우는 입력 ejb-jar 파일이 다른 애플리케이션 조립자에 의해 만들어진 경우에 발생한다.)
빈 제공자에 의해 만들어진 배치 디스크립터는 XML 형식을 지켜야 하며, 곧 설명될 DTD의 관점에서 유효해야 한다. 배치 디스크립터의 내용은 DTD 주석과 그 외 스펙에서 설명된 의미를 따라야 한다. 배치 디스크립터는 다음의 문장을 통해 DTD를 참고해야 한다.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

## 컨테이너 제공자의 책임

✍✍ 컨테이너 제공자는 XML 배치 디스크립터에 포함된 정보를 읽고 가져오는 (import) 툴을 제공한다.

## 배치 디스크립터 DTD

이 절에서는 EJB 1.1 배치 디스크립터를 위한 XML DTD를 정의한다. DTD의 설명은 DTD 메커니즘으로 쉽게 표현할 수 없는 문법이나 의미상의 부가적인 요구사항을 기술한다.

XML 엘리먼트의 내용은 보통 대소문자를 구별한다. 예를들어

```
<reentrant>True</reentrant>
```

위의 예제가 바른 예이며 다음의 예는 사용하지 않는 것이 좋다.

```
<reentrant>true</reentrant>
```

모든 유효한 ejb-jar 배치 디스크립터는 다음의 DOCTYPE 선언을 포함해야 한다.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

**<!ELEMENT assembly-descriptor (security-role*, method-permission*, container-transaction*)>**

assembly-descriptor 엘리먼트는 애플리케이션 조립 정보를 포함한다.

포함되는 모든 파트들은 선택 사항이므로 해당 파트가 비어있다면 그 파트는 생략된 것이다.

배치 디스크립터에서 ejb-jar 파일 생산자를 위해 assembly-descriptor를 제공하는 것은 선택 사항이다.

ejb-jar 엘리먼트에서 사용된다.

**<!ELEMENT cmp-field (description?, field-name)>**

cmp-field 엘리먼트는 컨테이너 관리 필드를 기술한다. 필드 엘리먼트는 필드의 선택적인 디스크립션이나 필드의 이름을 포함한다.

entity 엘리먼트에서 사용된다.

**<!ELEMENT container-transaction (description?, method+, trans-**

**attribute)>**

✍ container-transaction 엘리먼트는 어떻게 컨테이너가 엔터프라이즈 빈의 메소드 호출에 대한 트랜잭션 범위(scope)를 관리해야 하는지를 기술한다. 트랜잭션 속성은 모든 기술된 메소드에 대해 적용된다.

assembly-descriptor 엘리먼트에서 사용된다.

**<!ELEMENT description (#PCDATA)>**

description 엘리먼트는 부모 엘리먼트를 설명하는 텍스트를 제공하기 위하여 ejb-jar 파일 생산자에 의해서 사용된다.

cmp-field, container-transaction, ejb-jar, entity, env-entry, ejb-ref, method, method-permission, resource-ref, secu-rity-role, security-role-ref, 그리고 session 엘리먼트에서 사용된다.

**<!ELEMENT display-name (#PCDATA)>**

display-name 엘리먼트는 툴에 의해 보여지기 위한 짧은 이름을 포함한다.

ejb-jar, session, entity 엘리먼트에서 사용된다.

**<!ELEMENT ejb-class (#PCDATA)>**

ejb-class 엘리먼트는 엔터프라이즈 빈 클래스의 이름을 포함한다.

entity 와 session 엘리먼트에서 사용된다.

**<!ELEMENT ejb-client-jar (#PCDATA)>**

선택 사항인 ejb-client-jar 엘리먼트는 ejb-jar 파일 내의 엔터프라이즈 빈에 접근하기 위해 클라이언트 프로그램에 필요한 클래스 파일들을 포함하고 있는 JAR 파일을 기술한다. 배치자는 ejb-client JAR파일을 클라이언트의 클래스로더(class-loader)가 접근할 수 있도록 한다.

ejb-jar 엘리먼트에서 사용된다.

**예 :** <ejb-client-jar>employee_service_client.jar</ejb-client-jar>

**<!ELEMENT ejb-jar (description?, display-name?, small-icon?, large-icon?, enterprise-beans, assembly-descriptor?, ejb-client-jar?)>**

✍ ejb-jar 엘리먼트는 EJB 배치 디스크립터의 루트(root) 엘리먼트이다.

**<!ELEMENT ejb-link (#PCDATA)>**

ejb-link 엘리먼트는 ejb-jar 파일에서 다른 엔터프라이즈 빈과 연결되는 EJB 레퍼런스를 기술하는 ejb-ref 엘리먼트에서 사용된다.

ejb-link 엘리먼트의 값은 반드시 동일한 J2EE 애플리케이션 유닛 안에 있는 동일한 ejb-jar 파일 혹은 다른 ejb-jar 파일 내부의 엔터프라이즈 빈의 ejb-name이어야 한다.

ejb-ref 엘리먼트에서 사용된다.

**예:** <ejb-link>EmployeeRecord</ejb-link>

**<!ELEMENT ejb-name (#PCDATA)>**

ejb-name 엘리먼트는 엔터프라이즈 빈의 이름을 기술한다. 이 이름은 ejb-jar 파일의 배치 디스크립터에서 엔터프라이즈 빈의 이름을 명명하기 위해서 ejb-jar 파일 생산자가 할당된 이름이다. 이름은 같은 ejb-jar 파일의 엔터프라이즈 빈들의 이름들 사이에서 반드시 유일해야 한다.

엔터프라이즈 빈 코드는 이름과 상관이 없다. 그러므로 이름은 애플리케이션 조립 과정 중에도 엔터프라이즈 빈의 기능을 정지하지 않고 변경될 수 있다.

배치 디스크립터의 ejb-name과 배치자가 엔터프라이즈 빈의 홈에 할당할 JNDI 이름과의 구조적인 연관성은 없다.

이름은 반드시 NMTOKEN에 대한 사전적인 법칙을 따라야 한다.

entity, method, 그리고 session 엘리먼트에서 사용된다.

**예 :** <ejb-name>EmployeeService</ejb-name>

**<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home, remote, ejb-link?)>**

ejb-ref 엘리먼트는 다른 엔터프라이즈 빈의 홈에 대한 레퍼런스의 선언에 사용된다.

선택 사항인 ejb-link 엘리먼트는 참조되는 엔터프라이즈 빈을 기술하는데 사용된다. 보통 조립된 애플리케이션을 포함한 ejb-jar 파일에서 사용된다.

entity 와 session 엘리먼트에서 사용된다.

**<!ELEMENT ejb-ref-name (#PCDATA)>**

ejb-ref-name 엘리먼트는 EJB 참조의 이름을 포함한다. ‘ ejb/ ’ 로 접두사를 붙이는 것이 권고사항이다.

ejb-ref 엘리먼트에서 사용된다.

**예 :** <ejb-ref-name>ejb/Payroll</ejb-ref-name>

**<!ELEMENT ejb-ref-type (#PCDATA)>**

ejb-ref-type 엘리먼트는 참조되는 엔터프라이즈 빈의 요구되는 타입을 포함한다.

ejb-ref-type 엘리먼트는 다음 중 하나의 형태를 띄고 있어야 한다.

<ejb-ref-type>Entity</ejb-ref-type>

<ejb-ref-type>Session</ejb-ref-type>

ejb-ref 엘리먼트에서 사용된다.

**<!ELEMENT enterprise-beans (session | entity)+>**

enterprise-beans 엘리먼트는 하나, 그 이상의 엔터프라이즈 빈의 선언을 포함한다.

**<!ELEMENT entity (description?, display-name?, small-icon?, large-icon?, ejb-name, home, remote, ejb-class, persistence-type, prim-key-class, reentrant, cmp-field*, primkey-field?, env-entry*, ejb-ref*, security-role-ref*, resource-ref*)>**

entity 엘리먼트는 엔티티 빈을 선언한다. 선택 사항인 primkey-field는 엔티티의 persistency-type이 Container일 경우 디스크립터에 나타날 수 있다. 그외 엘리먼트 들은 생략 가능 하다는 관점에서 보면 모두 선택 사항이다.

만약 엔티티 빈의 persistence-type이 Container라면 적어도 하나의 cmp-field 엘리먼트는 디스크립터에 존재해야 한다. 만약 엔티티 빈의 persistence-type이 Bean이라면 cmp-field 엘리먼트는 존재할 수 없다.

enterprise-beans 엘리먼트에서 사용된다.

**<!ELEMENT env-entry (description?, env-entry-name, env-entry-type, env-entry-value?)>**

env-entry 엘리먼트는 엔터프라이즈 빈의 환경 엔트리에 대한 선언을 포함한다.

entity 와 session 엘리먼트에서 사용된다.

**<!ELEMENT env-entry-name (#PCDATA)>**

env-entry-name 엘리먼트는 엔터프라이즈 빈의 환경 엔트리의 이름을 포함한다.

env-entry 엘리먼트에서 사용된다.

예 : <env-entry-name>minAmount</env-entry-name>

**<!ELEMENT env-entry-type (#PCDATA)>**

env-entry-type 엘리먼트는 엔터프라이즈 빈의 코드에 의해 요구되는 환경 엔트리 값의 적합한 자바 타입을 포함한다.

env-entry 엘리먼트에서 사용된다.

예 : <env-entry-type>java.lang.Boolean</env-entry-type>

**<!ELEMENT env-entry-value (#PCDATA)>**

env-entry-value 엘리먼트는 엔터프라이즈 빈의 환경 엔트리의 값을 포함한다.

env-entry 엘리먼트에서 사용된다.

예 : <env-entry-value>100.00</env-entry-value>

**<!ELEMENT field-name (#PCDATA)>**

field-name 엘리먼트는 컨테이너 관리 필드의 이름을 기술한다. 이름은 엔터프라이즈 빈 클래스나 그 수퍼 클래스에서 반드시 public 필드여야 한다.

cmp-field 엘리먼트에서 사용된다.

예 : <field-name>firstName</field-Name>

**<!ELEMENT home (#PCDATA)>**

home 엘리먼트는 엔터프라이즈 빈의 홈 인터페이스의 이름을 포함한다.

ejb-ref, entity, session 엘리먼트에서 사용된다.

예 : <home>com.aardvark.payroll.PayrollHome</home>

**<!ELEMENT large-icon (#PCDATA)>**

large-icon 엘리먼트는 큰 (32 x 32) 아이콘 이미지 파일의 이름을 포함한다. 파일 이름은 ejb-jar 파일내의 상대경로로 표시된다.

이미지 파일은 반드시 JPEG나 GIF 포맷이어야 하며 파일 이름은 '.jpg'나 '.gif'로 끝나야 한다. 아이콘은 툴에 의해 사용된다.

예 : <large-icon>employee-service-icon32x32.jpg</large-icon>

**<!ELEMENT method (description?, ejb-name, method-intf?, method-name, method-params?)>**

method 엘리먼트는 엔터프라이즈 빈의 홈 인터페이스 또는 리모트 인터페이스의 메소드나 메소드의 집합을 나타내기 위해 사용된다. ejb-name 엘리먼트는 배치 디스크립터에 선언된 엔터프라이즈 빈 중 하나의 이름이어야 한다.

다음은 method 엘리먼트 문법의 세가지 스타일이다.

✍ 이 스타일은 기술된 엔터프라이즈 빈의 홈 인터페이스와 리모트 인터페이스의 모든 메소드를 언급하는데 사용된다.

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>*</method-name>
</method>
```

✍✍ 이 스타일은 기술된 엔터프라이즈 빈의 기술된 메소드를 언급하기 위해 사용된다. 만약 오버로딩된 같은 이름의 메소드가 여러 개 있다면, 이 스타일에서는 오버로딩된 이름을 갖는 모든 메소드들을 언급할 것이다.

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
</method>
```

✍✍ 이 스타일은 오버로딩된 이름을 가진 메소드들 중 하나의 메소드를 언급한 것이다. PARAM-1 부터 PARAM-n 은 메소드의 입력 파라미터의 자바 타입이다. 배열은 array 엘리먼트의 타입으로 기술된다.

method-permission 과 container-transaction 엘리먼트에서 사용된다.

예 :

다음 method 엘리먼트는 EmployeeService 빈의 홈 인터페이스와 리모트 인터페이스의 모든 메소드들을 언급한다.

```
<method>
  <ejb-name>EmployeeService</ejb-name>
  <method-name>*</method-name>
</method>
```

다음 method 엘리먼트는 EmployeeService 빈의 홈 인터페이스의 모든 create 메소드를 언급한 것이다.

```
<method>
  <ejb-name>EmployeeService</ejb-name>
  <method-name>create</method-name>
</method>
```

다음의 method 엘리먼트는 EmployeeService 빈의 홈 인터페이스의 create(String firstName, String LastName) 메소드를 언급한 것이다.

```
<method>
  <ejb-name>EmployeeService</ejb-name>
  <method-name>create</method-name>
  <method-params>
  <method-param>java.lang.String</method-param>
  <method-param>java.lang.String</method-param>
  </method-params>
```

```
</method>
```

**<!ELEMENT method-intf (#PCDATA)>**

method-intf 엘리먼트는 리모트 인터페이스와 홈 인터페이스에서 같은 이름과 모양을 가진 메소드드을 method 엘리먼트가 구별할 수 있게 해준다.

method-intf 엘리먼트는 다음 중 하나의 형식을 취해야 한다.

```
<method-intf>Home</method-intf>
<method-intf>Remote</method-intf>
```

method 엘리먼트에서 사용된다.

**<!ELEMENT method-name (#PCDATA)>**

method-param 엘리먼트는 엔터프라이즈 빈 메소드의 이름을 포함하거나 별표(asterisk : *)문자를 포함한다. 별표문자는 엘리먼트가 엔터프라이즈 빈의 리모트 인터페이스와 홈 인터페이스의 모든 메소드를 표시할 때 사용된다.

method 엘리먼트에서 사용된다.

**<!ELEMENT method-param (#PCDATA)>**

method-param 엘리먼트는 메소드 파라미터의 자바 타입 이름을 포함한다.

method-params 엘리먼트에서 사용된다.

**<!ELEMENT method-params (method-param*)>**

method-params 엘리먼트는 메소드 파라미터들의 자바 타입 이름들의 리스트를 포함한다.

method 엘리먼트에서 사용된다.

**<!ELEMENT method-permission (description?, role-name+, method+)>**

method-permission 엘리먼트는 하나 또는 그 이상의 엔터프라이즈 빈 메소드를 호출할 수 있는 보안 역할을 기술한다. method-permission 엘리먼트에서 사용된 보안 역할은 배치 디스크립터의 security-role 엘리먼트에 정의 되어야 하고 메소드는 엔터프라이즈 빈의 홈 인터페이스나 리모트 인터페이스에 정의된 메소드여야 한다.

assembly-descriptor 엘리먼트에서 사용된다.

**<!ELEMENT persistence-type (#PCDATA)>**

persistence-type 엘리먼트는 엔티티 빈의 퍼시스턴스 관리 타입을 기술한다.

persistence-type 엘리먼트는 다음 중 하나의 형식을 취해야 한다.

<persistence-type>Bean</persistence-type>

<persistence-type>Container</persistence-type>

entity 엘리먼트에서 사용된다.

**<!ELEMENT prim-key-class (#PCDATA)>**

prim-key-class 엘리먼트는 엔티티 빈의 프라이머리 키 클래스의 이름을 포함한다.

만약 프라이머리 키 클래스의 정의가 배치시까지 연기되면 prim-key-class 엘리먼트는 java.lang.Object 를 기술해야 한다.

entity 엘리먼트에서 사용된다.

예 :

<prim-key-class>java.lang.String</prim-key-class>

<prim-key-class>com.wombat.empl.EmployeeID</prim-key-class>

<prim-key-class>java.lang.Object</prim-key-class>

**<!ELEMENT primkey-field (#PCDATA)>**

primkey-field 엘리먼트는 컨테이너 관리 퍼시스턴스를 가지는 엔티티 빈의 프라이머리 키 필드의 이름을 기술하는데 사용된다.

primkey-field는 cmp-field 엘리먼트에 선언된 필드 중 하나여야 하고, 필드의 타입은 프라이머리 키 타입과 같아야 한다.

entity 엘리먼트에서 사용된다.

예 : <primkey-field>EmployeeId</primkey-field>

**<!ELEMENT reentrant (#PCDATA)>**

reentrant 엘리먼트는 엔티티 빈이 재진입이 가능한지 여부를 기술한다.

reentrant 엘리먼트는 다음 중 하나의 형식을 가져야 한다.

<reentrant>True</reentrant>

<reentrant>False</reentrant>

entity 엘리먼트에서 사용된다.

**<!ELEMENT remote (#PCDATA)>**

remote 엘리먼트는 엔터프라이즈 빈의 리모트 인터페이스의 이름을 포함한다.

ejb-ref, entity, session 엘리먼트에서 사용된다.

**<!ELEMENT res-auth (#PCDATA)>**

res-auth 엘리먼트는 엔터프라이즈 빈 코드가 리소스 매니저에게 프로그래밍적으로 서명할 것인지 아니면 컨테이너가 빈을 대신해서 서명할 것인지를 기술한다. 후자의 경우 컨테이너는 배치자에게 제공받은 정보를 이용한다.

이 엘리먼트의 값은 다음 둘 중 하나여야 한다.

```
<res-auth>Application</res-auth>
<res-auth>Container</res-auth>
```

resource-ref 엘리먼트에서 사용된다.

**<!ELEMENT res-ref-name (#PCDATA)>**

res-ref-name 엘리먼트는는 리소스 매니저 컨넥션 팩토리 레퍼런스(resource manager connection factory reference)의 이름을 기술한다.

resource-ref 엘리먼트에서 사용된다.

**<!ELEMENT res-type (#PCDATA)>**

res-type 엘리먼트는 데이터 소스의 타입을 기술한다. 타입은 데이터 소스에 의해 구현되는 자바 인터페이스(또는 클래스)에 의해 기술된다.

resource-ref 엘리먼트에서 사용된다.

**<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth)>**

resource-ref 엘리먼트는 외부 자원을 참조하는 엔터프라이즈 빈의 선언을 포함한다.

entity 와 session 엘리먼트에서 사용된다.

예 :

```
<resource-ref>
  <res-ref-name>EmployeeAppDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

**<!ELEMENT role-link (#PCDATA)>**

role-link 엘리먼트는 보안 역할 참조를 정의된 보안 역할과 연결하는데 사용된다. role-link 엘리먼트는 반드시 security-role 엘리먼트에 정의된 보안 역할 중 하나의 이름을 포함해야한다.

security-role-ref 엘리먼트에서 사용된다.

**<!ELEMENT role-name (#PCDATA)>**

role-name 엘리먼트는 보안 역할의 이름을 포함한다. 이름은 반드시 NMTOKEN을 위한 어휘 법칙(lexical rules)을 따라야 한다.

method-permission, security-role, security-role-ref 엘리먼트에서 사용된다.

**<!ELEMENT security-role (description?, role-name)>**

security-role 엘리먼트는 보안 역할의 정의를 포함한다.

assembly-descriptor 엘리먼트에서 사용된다.

```
<security-role>
  <description>
    This role includes all employees who are authorized
    to access the employee service application.
  </description>
  <role-name>employee</role-name>
</security-role>
```

**<!ELEMENT security-role-ref (description?, role-name, role-link?)>**

security-role-ref 엘리먼트는 엔터프라이즈 빈 코드에서 보안 역할 레퍼런스의 선언을 포함한다.

role-name 엘리먼트의 값은 EJBContext.isCallerInRole(String roleName) 메소드의 파라미터 값에서 사용되는 String 이어야 한다. role-link 엘리먼트의 값은 security-role 엘리먼트에 정의된 보안 역할 중 하나의 이름이어야 한다.

entity 와 session 엘리먼트에서 사용된다.

**<!ELEMENT session-type (#PCDATA)>**

session-type 엘리먼트는 세션 빈이 상태유지 세션 빈 인지 무상태 세션 빈 인지를 기술한다.

session-type 엘리먼트는 다음 둘 중 하나의 형식을 가져야 한다.

```
<session-type>Stateful</session-type>
<session-type>Stateless</session-type>
```

**<!ELEMENT session (description?, display-name?, small-icon?, large-icon?, ejb-name, home, remote, ejb-class, session-type, transaction-type, env-entry*, ejb-ref*, security-role-ref*, resource-ref*)>**

session 엘리먼트는 세션 빈을 선언한다. 필수 요소가 아닌 엘리먼트 들은 생략 가능 하다는 관점에서 보면 모두 선택 사항이다.

enterprise-beans 엘리먼트에서 사용된다.

**<!ELEMENT small-icon (#PCDATA)>**

small-icon 엘리먼트는 작은 (16 x 16) 아이콘 이미지 파일의 이름을 포함한다. 파일 이름은 ejb-jar 파일 내의 상대 경로로 표시된다.

이미지 파일은 반드시 JPEG나 GIF 포맷이어야 하며 파일 이름은 '.jpg'나 '.gif'로 끝나야 한다. 아이콘은 툴에 의해 사용된다.

예 : <small-icon>employee-service-icon16x16.jpg</small-icon>

**<!ELEMENT transaction-type (#PCDATA)>**

transaction-type 엘리먼트는 엔터프라이즈 빈의 트랜잭션 관리 타입을 기술한다.

transaction-type 엘리먼트는 다음 중 하나의 형식을 가져야 한다.

```
<transaction-type>Bean</transaction-type>
<transaction-type>Container</transaction-type>
```

**<!ELEMENT trans-attribute (#PCDATA)>**

trans-attribute 엘리먼트는 컨테이너가 메소드 호출을 엔티티 빈의 비즈니스 메소드로 위임할 때 트랜잭션 경계를 어떻게 관리해야 할것인가를 기술한다.

trans-attribute 엘리먼트의 값은 다음 중 하나가 되어야 한다.

```
<trans-attribute>NotSupported</trans-attribute>
<trans-attribute>Supports</trans-attribute>
<trans-attribute>Required</trans-attribute>
<trans-attribute>RequiresNew</trans-attribute>
<trans-attribute>Mandatory</trans-attribute>
<trans-attribute>Never</trans-attribute>
```

container-transaction 엘리먼트에서 사용된다.

✍✍ ID 메커니즘은 분리된 파일에 비표준 정보를 저장하고, 이러한 툴 의존적인 파일들을로부터 표준 배치 디스크립터의 정보를 쉽게 참조할 수 있도록 툴이 부가적인 배치 정보를 생성할 수 있도록 한다.

EJB 아키텍쳐에서는 툴이 EJB 배치 디스크립터에 비표준 정보를 추가할 수 없다.

```
<!ATTLIST assembly-descriptor id ID #IMPLIED>
<!ATTLIST cmp-field id ID #IMPLIED>
<!ATTLIST container-transaction id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-class id ID #IMPLIED>
<!ATTLIST ejb-client-jar id ID #IMPLIED>
<!ATTLIST ejb-jar id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-name id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST enterprise-beans id ID #IMPLIED>
<!ATTLIST entity id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST field-name id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST method id ID #IMPLIED>
<!ATTLIST method-intf id ID #IMPLIED>
<!ATTLIST method-name id ID #IMPLIED>
<!ATTLIST method-param id ID #IMPLIED>
<!ATTLIST method-params id ID #IMPLIED>
<!ATTLIST method-permission id ID #IMPLIED>
<!ATTLIST persistence-type id ID #IMPLIED>
<!ATTLIST prim-key-class id ID #IMPLIED>
<!ATTLIST primkey-field id ID #IMPLIED>
<!ATTLIST reentrant id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
```

```
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST session-type id ID #IMPLIED>
<!ATTLIST session id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST transaction-type id ID #IMPLIED>
<!ATTLIST trans-attribute id ID #IMPLIED>
```

# Descriptor 작성하기

## 실제적인 배치 디스크립터 작성 예제

여기서 예제로 살펴볼 컨테이너 관리 엔티티 빈은 고용자를 모델링한 간단한 빈이다. 이번 예제를 통해 실제적으로 DTD를 적용하여 배치 디스크립터를 작성해 보자.

## 소스코드

다음은 엔티티 빈의 소스코드이다.

✍✍ 홈 인터페이스

```
import javax.ejb.*;
import java.rmi.*;
import java.util.*;

public interface EmpContainerHome extends EJBHome
{
   public EmpContainer create(int empno, String ename,
String job, int mgr, Date hiredate,
int sal, int comm, int deptno)
throws CreateException, RemoteException;
   public EmpContainer findByPrimaryKey(Integer empno)
throws FinderException, RemoteException;
   public Enumeration findByDeptno(int deptno)
throws FinderException, RemoteException;
}
```

<소스> EmpContainerHome.java

✍✍ 리모트 인터페이스

```
import javax.ejb.*;
```

```
import java.rmi.*;

public interface EmpContainer extends EJBObject
{
   public String getEname() throws RemoteException;
   public int getEmpno() throws RemoteException;
   public String getJob() throws RemoteException;
   public int getMgr() throws RemoteException;
   public java.util.Date getHiredate() throws RemoteException;
   public int getSal() throws RemoteException;
   public int getComm() throws RemoteException;
   public int getDeptno() throws RemoteException;
}
```

<소스> EmpContainer.java

✍✍ **빈 클래스**

```
import javax.ejb.*;

public class EmpContainerBean implements EntityBean
{
   public String ename;
   public int    empno;
   public String job;
   public int    mgr;
   public java.sql.Date hiredate;
   public int    sal;
   public int    comm;
   public int    deptno;

   EntityContext ctx;

   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void ejbRemove()  {}
   public void setEntityContext(EntityContext ctx)    {
      this.ctx = ctx;
   }
   public void ejbLoad() {}
```

```
    public void ejbStore() {}
    public void unsetEntityContext() {}

    public Integer ejbCreate(int empno, String ename, String job,
int mgr, java.util.Date hiredate, int sal,
int comm, int deptno) {
        this.empno = empno;
        this.ename = ename;
        this.job = job;
        this.mgr = mgr;
        this.hiredate = new java.sql.Date(hiredate.getTime());
        this.sal = sal;
        this.comm = comm;
        this.deptno = deptno;
        return null;
    }
    public void ejbPostCreate(int empno, String ename,
String job, int mgr,
java.util.Date hiredate, int sal,
int comm, int deptno)  {}
    public String getEname()
    {
        return ename;
    }
    public int getEmpno()
    {
        return empno;
    }
    public String getJob()
    {
     +   return job;
    }
    public int getMgr()
    {
        return mgr;
    }
    public java.util.Date getHiredate()
    {
        return new java.util.Date(hiredate.getTime());
```

```
    }
    public int getSal()
    {
        return sal;
    }
    public int getComm()
    {
        return comm;
    }
    public int getDeptno()
    {
        return deptno;
    }
}
```

<소스> EmpContainerBean.java

✍✍ **배치 디스크립터**

다음은 컨테이너 관리 빈인 EmpContainer 빈의 배치 디스크립터를 작성한 예이다.

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

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
   <enterprise-beans>
     <entity>
    <ejb-name>EmpContainer</ejb-name>
    <home>EmpContainerHome</home>
    <remote>EmpContainer</remote>
    <ejb-class>EmpContainerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>EmpPK</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field>
      <field-name>deptno</field-name>
    </cmp-field>
    <cmp-field>
```

```
      <field-name>mgr</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>comm</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>hiredate</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>job</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>empno</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>sal</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>ename</field-name>
    </cmp-field>
     </entity>
   </enterprise-beans>
   <assembly-descriptor>
    <container-transaction>
    <method>
      <ejb-name>EmpContainer</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
     </container-transaction>
   </assembly-descriptor>
</ejb-jar>
```

<소스> ejb-jar.xml