<JSTORM>

EJB의 트랜잭션

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

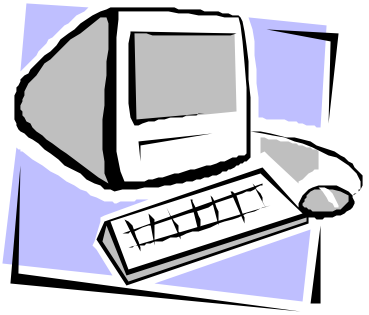
Document Information

Document title:	EJB의 트랜잭션
Document file name:	EJBtransection_pjh_EJB_jstorm_1_final
Revision number:	<1.0>
Issued by:	박지훈 (outinsider@orgio.net)
Issue Date:	<2001/6/21 >
Status:	Final

Content Information

Audience	JAVA개발자 및 컴퓨터 관련 전공자
Abstract	EJB 1.1의 트랜잭션에 대한 전반적인 개요를 설명하였다. 본 내용은 2001년 6월에 출간된 대청미디어의 "엔터프라이즈 자바빈즈" 서적의 "4.5 트랜잭션"의 초반기 원고를 그대로 인용하였다.
Reference	1) [Bulka] 엔터프라이즈 자바빈즈(Jstorm 박지훈 외 5인), 대청미디어, 2001 2) 인터넷의 각종 EJB 트랜잭션 자료 3) 선사의 J2EE 자료
Benchmark information	본 원고가 속한 책의 자세한 정보는 아래와 같다. 1) 라이코스 서적 http://book.lycos.co.kr/common/bookinfo/bookinfo.asp?isbn=ISBN89-87939-51-0 본 원고는 초반기 원고이므로 정식으로 출판된 책의 내용보다 오탈자가 많을 수 있다.

Table of Contents



4.5 트랜잭션(Transaction)

엔터프라이즈 자바빈즈의 핵심 특징 중의 하나가 분산 트랜잭션을 지원하는 것이다. EJB 어플리케이션 개발자는 여러 사이트에 분산되어 있는 데이터베이스들간의 트랜잭션들을 쉽게 구현할 수 있다.

4.5.1 트랜잭션이란?

1) 트랜잭션의 개요

트랜잭션이란 ACID(Atomicity, Consistency, Isolation, Durability) 속성을 만족해야 하는 일의 최소 단위이다. ACID속성은 다음과 같다.

- Atomicity : 어떤 일이 완전히 수행되거나 또는 전혀 수행되지 않은 상태로 되어야 함. 즉 "All or Nothing"임
- Consistency : 트랜잭션이 끝난 후 데이터들은 일관성을 유지해야 함.
- Isolation : 여러 트랜잭션이 동시에 수행될 때 한 트랜잭션이 다른 트랜잭션이 사용중인 불완전한 데이터를 공동으로 사용하는것을 방지해야 함. 여러 트랜잭션들이 순서대로 실행되게 하는 일렬화(Serialization)를 제공해야 함.
- Durability : 트랜잭션 수행결과는 반드시 2차 저장소에 영구적으로 반영되어야 함.

예를 들어 은행업무시스템에서 "이체한다"는 트랜잭션이다. "이체한다"는 "1.내 계좌에서 이체할 금액을 뺀다"와 "2.이체할 계좌로 전송한다"라는 서브태스크로 쪼개질 수 있다. "이체한다"는 일이 왜 트랜잭션인지 ACID속성으로 설명해보자.

a. Atomicity

"이체한다"는 일은 일단 시작하면 완전히 끝이 나거나 아니면 시작하지 않은 원래 상태로 되어야한다. 예를 들어 "이체한다"는 일이 시작되어서 "1.내 계좌에서 이체할 금액을 뺀다."는 성공적으로 수행되었는데 시스템에 문제가 생겨서 "2.이체할 계좌로 전송한다"가 수행되지 않았다고 해보자. (네트웍 및 하드웨어, 소프트웨어의 다운 등이 시스템의 문제이다.) 그러면 내 계좌에서 돈이 빠져나갔지만 실제로 상대방 통장에 이체가 되지 않은 결과가 발생한다. 이럴 경우에는 "1.내 계좌에서 이체할 금액을 뺀다"를 취소하여 이체하기전의 원래 상태로 돌려야 한다.

b. Consistency

내 계좌의 금액과 이체할 계좌의 금액은 일관성이 있어야 한다. 즉, 내 계좌에서 돈이 A만큼 빠졌으면 이체할 계좌의 금액은 A만큼 증가해야 한다.

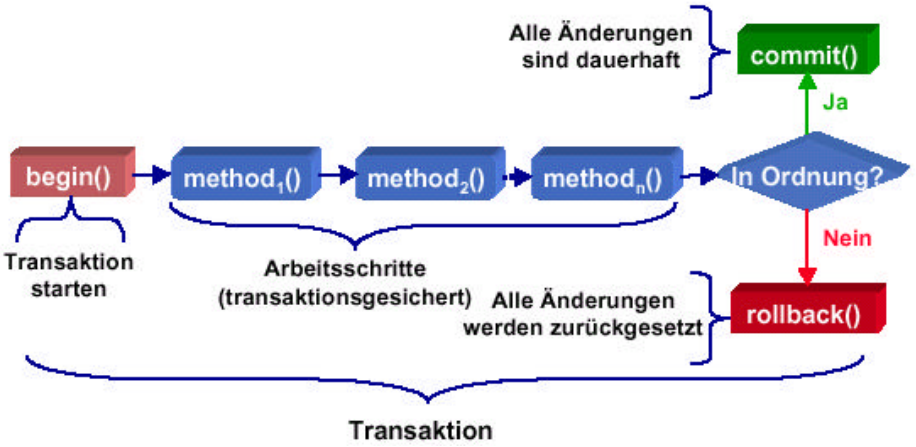
c. Isolation

내가 "이체한다"를 수행할 동안 동시에 다른 사람도 "이체한다"를 수행할 수 있다. 이때 데이터가 서로 공유될 수 있다. 예를 들어 내가 다른 사람의 계좌로 돈을 이체하는 순간, 동시에 다른 사람이 내 계좌로 돈을 이체할 수 있다. 이때 내 계좌의 금액이 서로 공유되는 것이다. 즉 내 계좌의 데이터가 두개의 "이체한다"라는 일에 의해 공유되어 사용되어지기 때문에 데이터의 일관성을 확보하기 위해서는 한 사람의 "이체한다"가 완전히 끝나고 다른 사람의 "이체한다"가 수행되어야 할 것이다.

d. Durability

"이체한다"는 일이 완전히 끝나면 그 결과인 내 예금 계좌와 이체할 계좌의 값이 데이터베이스에 반영되어야 한다. 즉, 내 예금계좌는 이체한 금액만큼 감소되어야 하고 이체할 계좌는 이체한 금액만큼 증가하도록 데이터베이스를 업데이트해야 한다.

이런 트랜잭션은 일의 시작의 "시작(begin)"과 일의 정상 종료시 "커미트(commit)", 비정상 종료시 "롤백(rollback)"이라는 상태가 있다. "시작"은 트랜잭션의 시작을 의미하고 "커미트"는 정상적으로 트랜잭션이 종료되었음을 의미하고 "롤백"은 트랜잭션이 정상적으로 종료되지 않았을 경우 원상태로 되돌림을 의미한다.



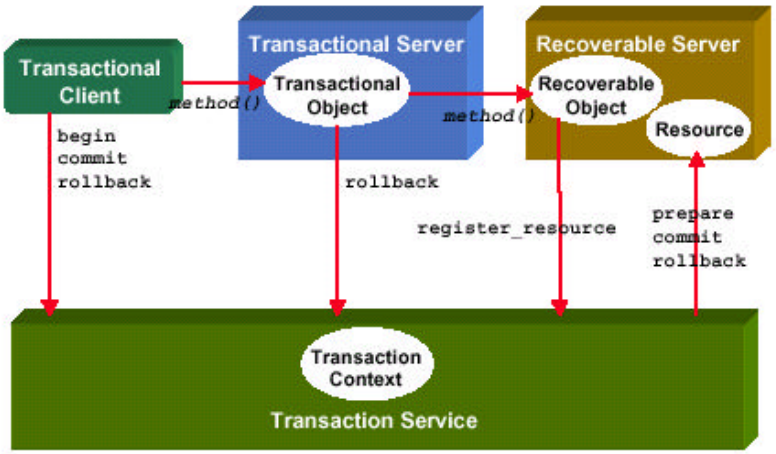
<그림 1> 트랜잭션 (Transaction starten->트랜잭션 시작,
Arbeitsschritte(...)->서브태스크들->Transaction->트랜잭션,
Alle Anderungen sind dauerhaft->트랜잭션 정상종료,
All Anderungen werden zur ...-> 트랜잭션 비정상 종료,
In Ordnung->정상종료?, Ja->예, Nein->아니오)

기업용 어플리케이션 개발자들은 이런 트랜잭션들을 소프트웨어적으로 보장되도록 시스템을 개발해야 하는데 이것은 쉬운일이 아니다. 따라서 이런 일을 전담하는 시스템이 별도로 존재하여 널리 사용되었다. 이런 시스템을 TP모니터(TP Monitor)라고 한다.

2) EJB의 트랜잭션 모델

J2EE의 분산객체 트랜잭션은 COBA의 OTS(Object Transaction Service)를 따르는 JTS(Java Transaction Service)로 처리한다.

OTS는 CORBA가 제안한, 분산객체 트랜잭션의 표준안으로서 다음과 같은 구조이다.



<그림 2> OTS의 구조

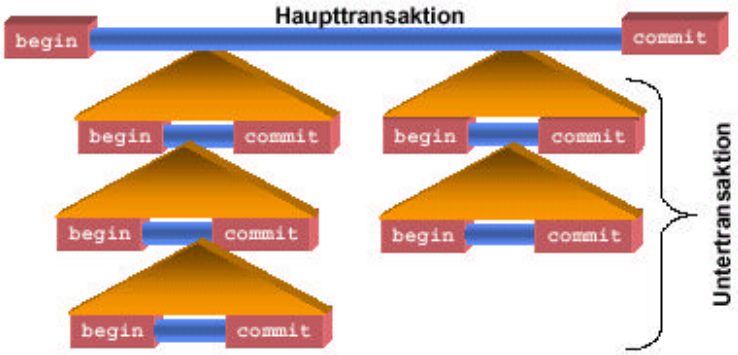
OTS는 X/Open의 DTP 표준을 따르며 분산 트랜잭션과 플랫 트랜잭션을 지원한다. JTS는

OTS를 자바언어로 바인딩(binding)한 것이다.

트랜잭션 모델은 대표적으로 플랫 트랜잭션과 네스티드 트랜재션(Nested Transaction)이 있다. EJB1.1이 필수적으로 지원하는 것은 플랫 트랜잭션이며 네스티드 트랜잭션은 지원하지 않는다.
플랫 트랜잭션(Flat Transaction)이란 한 개의 트랜잭션내에 또 다른 트랜잭션이 포함되지 않고 오직 1개의 트랜잭션만을 수행하는 것이다. 여러 개의 트랜잭션이 동시에 발생되어도 아래 그림과 같이 논리적 시간순서대로 차례차례 실행된다.

<그림 3> 플랫 트랜잭션 (transaktion->트랜잭션)

네스티드 트랜잭션은 좀 복잡한 형태로서 한 트랜잭션(메인 트랜잭션)안에 또 다른 트랜잭션(서브 트랜잭션)이 포함될 수 있다. 포함되어 있는 서브 트랜잭션에서 문제가 발생했을 경우 전체 트랜잭션을 롤백하지 않고 자신만 롤백할 수 있기 때문에 플랫 트랜잭션보다 효율적이다.



<그림 4> 네스티드 트랜잭션 (Haupttransaction->메인 트랜잭션, Untertransaktion->서브트랜잭션)

예를 들어 플랫 트랜잭션인 경우 어떤 트랜잭션이 100개 정도되는 서브태스크로 실행된다고 해보자.“1->2->3->4>…->100”을 실행하다가 99번째 서브태스크를 실행하다 문제가 발생하면 1부터 99까지 실행했던 일을 모두 취소하고 원상태로 되돌려야 한다. 그렇지만 네스티드 트랜잭션인 경우 서브 태스크들을 독립적인 트랜잭션으로 할 수 있기 때문에 99번째 서브태스크가 문제가 생기면 그 태스크만 롤백하고 그 이후에 다시 태스크를 진행할 수 있다.

EJB어플리케이션을 개발할 때 트랜잭션 처리, 즉 어떤 작업을 ACID 속성이 만족되도록 구현하는 것을 EJB컨테이너가 지원하기 때문에 개발자는 트랜잭션 처리에 비교적 자유롭다.

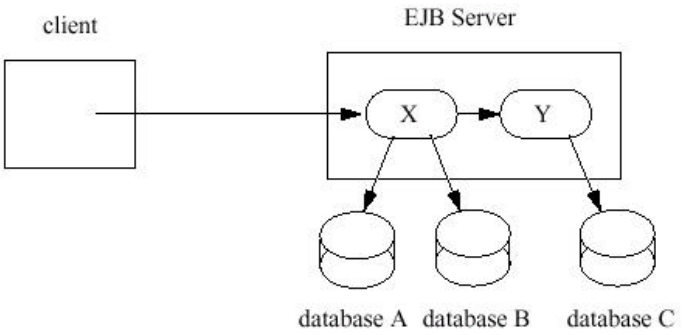
EJB 표준에 따르면 EJB컨테이너업체들은 반드시 JTS를 구현할 필요는 없다. 단지 JTA(Java Transaction API)인 javax.transaction,UserTransaction 인터페이스를 반드시 구현해야 한다. 즉. EJB에서 트랜잭션 내부 처리 방법은 EJB 컨테이너를 만드는 회사에게 전적으로 맡긴 셈이다.

4.5.2 EJB 트랜잭션 시나리오

EJB컨테이너는 분산객체 트랜잭션을 보장하는 막강한 트랜잭션 지원기능이 있다. 이런 트랜잭션 지원을 이용하여 다음과 같은 시나리오로 EJB어플리케이션을 개발할 수 있다.

가. 여러 개의 데이터베이스의 업데이트

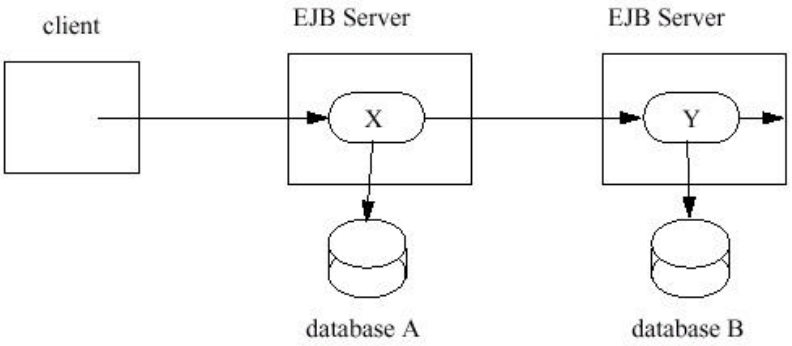
아래 그림처럼 한 트랜잭션내에서 여러 개의 데이터베이스에 있는 데이터들을 업데이트할 때 트랜잭션이 보장된다.



<그림 5> 여러 데이터베이스들과 연결된 트랜잭션

나. 여러 개의 EJB 서버들을 통한 데이터베이스들의 업데이트

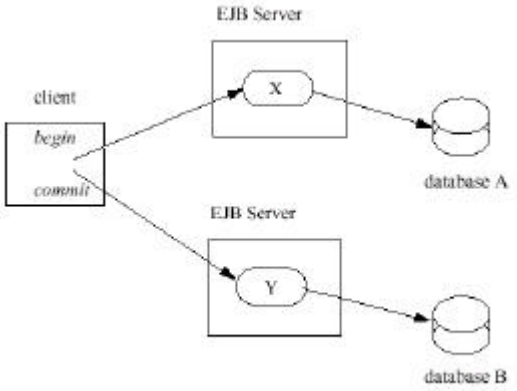
아래 그림과 같이 한 트랜잭션내에서 EJB서버가 설치된 여러 개의 사이트들의 데이터베이스들을 업데이트할 수 있다.



<그림 6> 분산된 EJB서버들에 걸친 트랜잭션 처리

다. 클라이언트 소스코드내에서 트랜잭션을 시작하며 관리

EJB 클라이언트 개발자들은 javax.transaction.UserTransaction 인터페이스를 사용하여 트랜잭션을 시작하며 관리할 수 있다. 이때 J2EE의 JNDI API를 사용하여 UserTransaction 객체를 확보할 수 있으며 이 객체의 메쏘드를 사용하여 트랜잭션의 시작, 커미트, 롤백 등을 할 수 있다. 하지만 초보자는 실수를 하기 쉬운 방법으로서 트랜잭션 전문가만 사용하도록 권고한다.

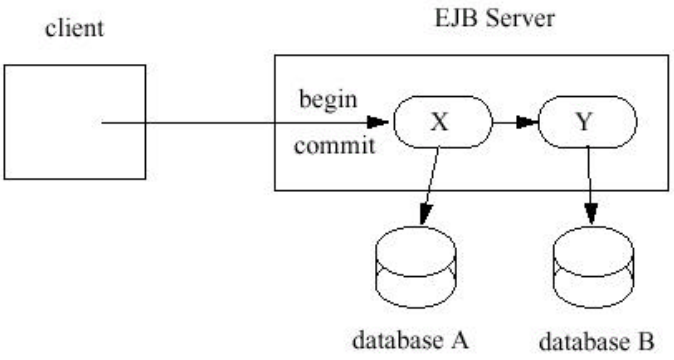


<그림 7> 클라이어트가 시작한 트랜잭션

라. EJB 컨테이너에 의해 트랜잭션이 자동으로 시작되며 관리

엔터프라이즈 빈 소스코드 내에 트랜잭션 관리 코드가 포함되지 않고 디플로이먼트할 때 6가지 트랜잭션 속성 중 하나가 선택된다. 클라이언트내에 트랜잭션 코드가 없어도 지정된 트랜잭션 속성대로 트랜잭션을 자동으로 관리한다. 이 방법을 가장 많이 사용한다.

아래 그림처럼 한 개의 트랜잭션내에 여러 개의 데이터베이스가 있어도 가능하다.



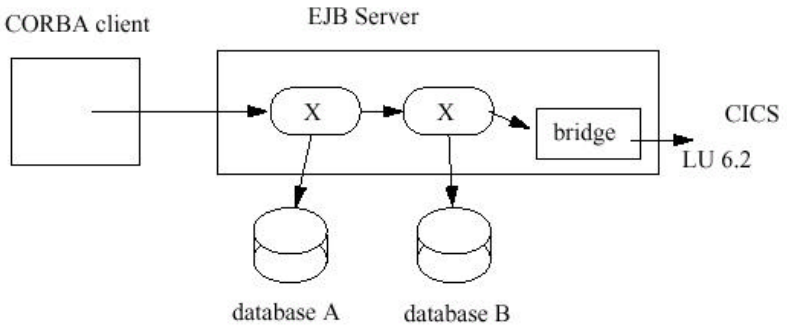
<그림 8> EJB 컨테이너가 자동으로 시작한 트랜잭션

마. 엔터프라이즈 빈 소스코드내에서 트랜잭션을 시작하고 관리

세션빈 내에서 javax.transaction.UserTransaction 을 사용하여 트랜잭션을 프로그래밍할 수 있다. 트랜잭션의 시작 및 커미트, 롤백을 직접 코딩한다. 반면 엔티티빈은 기본적으로 컨테이너가 트랜잭션을 관리하기 때문에 개발자가 직접 트랜잭션 관리 코드를 작성할 필요는 없다.

바. 자바가 아닌 클라이언트 또는 서버와의 호환성

CORBA가 실행가능한 EJB 컨테이너안의 엔터프라이즈빈을 다른 언어로 작성된 코바객체가 호출할 수 있다. 한 브릿지를 사용하여 엔터프라이즈빈이 다른 서버내의 컴포넌트를 호출할 수 있다. 다른 컴포넌트들과의 브릿지(bridge) 및 CORBA 실행가능한 EJB서버는 EJB 스펙범위 밖이다.



<그림 9> 코바와 다른 시스템과 연동하는 EJB 컴포넌트

4.5.3 EJB 트랜잭션 타입

트랜잭션은 일단 시작(begin)하면 커미트(Commit)되거나 롤백(Rollback) 되어야 한다. 트랜잭션을 지원하는 개발 플랫폼에서 트랜잭션을 처리하는 어플리케이션 개발자는 이런 트랜잭션들을 시작하고 커미트하고 롤백하는 주체(who)와 시기(when)를 잘 알아야 한다. 누가 트랜잭션을 시작, 커미트, 롤백을 하는지, 그리고 그것들이 발생되는 시간을 정확히 이해해야 완벽한 트랜잭션 어플리케이션을 만들 수 있다. 이것은 트랜잭션의 경계(boundary)라고 한다.

객체지향 시스템에서 트랜잭션의 단위는 객체의 메쏘드 자체이거나 메쏘드안의 일부분, 또는 여러 메쏘드 등이 될 수 있다. 예를 들어 "이체한다"는 트랜잭션의 단위가 "이체한다"는 메쏘드 자체가 될 수 있다. 또는 "내 계좌에서 이체할 금액을 뺀다."와 "이체할 계좌로 전송한다"들이 별개의 메쏘드로 구현되어 있을 경우 두개의 메쏘드가 하나의 트랜잭션 단위가 된다. 이런 트랜잭션 메쏘드들은 데이터베이스를 업데이트한다.

	한 메쏘드가 트랜잭션 단위일 경우	여러 메쏘드가 트랜잭션 단위일 경우

메쏘드 정의 예	이체한다() {
	내 계좌에서 이체할 금액을 뺀다;
	이체할 계좌로 전송한다;
	}
		내 계좌에서 이체할 금액을 뺀다() {
		…
		}
		이체할 계좌로 전송한다() {
		…
		}

트랜잭션 단위	tx.begin; // 트랜잭션 시작
	이체한다();
	tx.commit; // 트랜잭션 정상종료
		tx.begin; // 트랜잭션 시작
		내 계좌에서 이체할 금액을 뺀다();
		이체할 계좌로 전송한다();
		tx.commit; // 트랜잭션 정상종료

<표 1> 메쏘드와 트랜잭션 단위

일반적으로 관리하기 편하도록 한 개의 메쏘드를 트랜잭션 단위로 많이 사용한다.

트랜잭션을 처리하는 EJB 어플리케이션을 만들 때 두가지 경우가 있다. 첫번째로 개발자가 직접 트랜잭션 관리 코드를 작성하는 것과 두번째로 소스코드내에 트랜잭션 관리 코드가 없고 디플로이먼트 디스크립터안에 트랜잭션 타입을 지정(선언)하는 것이다. 후자일 경우는 EJB컨테이너가 자동으로 트랜잭션을 관리해준다. 전자를 "프로그래밍 트랜잭션 방식(programmign transaction type)", 후자를 "선언 트랜잭션 방식(declarative tran

saction type)"라고 부른다. 대부분 개발자는 후자의 방식을 선택하여 트랜잭션을 관리하는 것이 좋다. 전자처럼 개발자가 직접 트랜잭션 코드를 작성할 때 실수할 경우가 있겠지만 후자처럼 EJB컨테이너가 직접 트랜잭션을 관리할 때엔 그런 일은 없기 때문이다. 그렇지만 개발자가 직접 트랜잭션 관리 코드를 작성하는 것이 EJB컨테이너가 관리하는 트랜잭션보다 좀 더 유연하고 확장성있게 만들 수 있다. 수공예품이 공장에서 나오는 공예품보다 훨씬 더 정밀하게 만들 수 있는 것처럼 말이다.

"프로그래밍 트랜잭션 방식"인 경우 개발자가 트랜잭션의 시작과 커미트, 롤백을 프로그래밍으로 지시한다. 이때 소스코드내에 배치한 트랜잭션 코드에 따라 트랜잭션의 시작, 커미트, 롤백이 발생한다. 이때 자바의 트랜잭션 관리 API인 JTA(Java Transaction API)를 사용하여 코딩한다. 반면 "선언 트랜잭션 방식"은 디플로이먼트할 때 디플로이먼트 디스크립터안에서 트랜잭션 속성을 지시한다. 그러면 EJB 컨테이너가 트랜잭션의 시작과 커미트, 롤백을 지정된 트랜잭션 방식에 따라 자동으로 적당한 시기에 실행한다.

" 프로그래밍 트랜잭션 처리"는 또다시 2개의 종류로 나눠지는데 하나는 "클라이언트에 의한 트랜잭션관리"와 "엔터프라이즈 빈에 의한 트랜잭션관리"이다. 전자는 엔터프라이즈 빈을 호출하는 클라이언트, 후자는 서버에서 동작하는 엔터프라이즈 빈내에서 트랜잭션이 시작,커미트,롤백되는 것이다. 반면 "선언 트랜잭션 처리"는 "컨테이너에 의한 트랜잭션관리"이다.

프로그래밍 트랜잭션 방식	선언 트랜잭션 방식
클라이언트에 의한 트랜잭션 관리	컨테이너에 의한 트랜잭션 관리
엔터프라이즈 빈에 의한 트랜잭션관리	

<표 2> EJB의 트랜잭션 종류

EJB어플리케이션에서 자주 사용되는 트랜잭션 종류는 "컨테이너에 의한 트랜잭션 관리"가 가장 널리 쓰이며 그 다음은 "엔터프라이즈 빈에 의한 트랜잭션 관리"이다. 이 두 방식을 사용하려면반드시 디플로이먼트 디스크립터안에 두가지 방식 중 어떤 방식을 쓰는지 명시한 후 디플로이먼트해야 한다. 그래야 비로서 EJB컨테이너가 디플로이먼트된 엔터프라이즈 빈을 어떤 방식으로 트랜잭션을 관리하는지 결정한다.

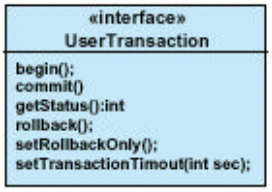
대부분 개발자들이 "컨테이너에 의한 트랜잭션 관리"로서 트랜잭션을 처리하기 때문에 초보 EJB어플리케이션 개발자는 다음 내용중에서 곧장 "컨테이너에 의한 트랜잭션 관리"를 봐도 무방하다.

4.5.4 클라이언트에 의한 트랜잭션 관리

엔터프라이즈 빈을 호출하는 클라이언트에서 javax.transaction.UserTransaction 이란 인

터페이스의 객체를 사용하여 트랜잭션의 시작, 커미트, 롤백을 지시하는 경우를 의미한다.

이 인터페이스는 선언된 메쏘드는 다음과 같다.



<그림 10> javax.transaction.UserTransaction 인터페이스

이 인터페이스를 사용하여 다음과 같은 일을 할 수 있다.

- 트랜잭션 시작,커미트,롤백을 수행한다.(begin(),commit(),rollback())
- 트랜잭션의 롤백을 강요한다. (setRollbackOnly())
- 트랜잭션의 상태정보를 얻는다. (getStatus())
- 트랜잭션의 타임아웃(timeout)을 지정한다. (setTransactionTimeout())

이 메쏘드들 중 begin()메쏘드가 트랜잭션의 시작, commit()메쏘드가 트랜잭션의 커미트, rollback()메쏘드가 트랜잭션의 롤백을 명령한다. 소스코드내의 begin()과 commit()사이가 트랜잭션 범위가 된다. 주의할 점은 트랜잭션 안에 java.sql.Connection의 commit(), rollback()메쏘드등을 호출하면 안된다는 것이다. 그것은 이미 JTA에 의해 관리되는 트랜잭션이 방해되기 때문이다.

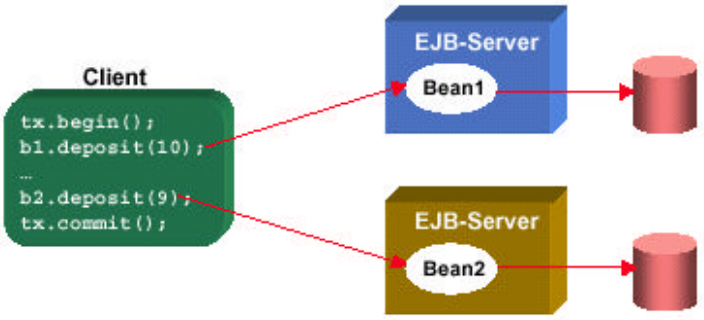
클라이언트가 UserTransaction객체를 얻기 위해서는 다음 예제처럼 JNDI를 사용한다.

```
java.util.Properties env = ...
Context ctx = new InitialContext(env);
tx=(javax.transaction.UserTransaction)ctx.lookup("javax.transaction.UserTra
nsaction");
tx.begin(); // 트랜잭션 시작
bean.deposit(100); // 엔터프라이즈 빈 호출, 여러 메쏘드가 들어올 수 있다.
tx.commit(); 또는 tx.rollback(); // 트랜잭션 종료
```

<예제 1> 웹로직의 "클라이언트에 의한 트랜잭션 관리"

아래 그림처럼 클라이언트에 의한 트랜잭션 관리는 주로 여러 개의 서버에 있는 엔터프

라이즈 빈을 호출할 때 그 호출이 한 트랜잭션내에 관리하고 싶은 경우에 사용한다.



<그림 11> 한 트랜잭션내에 여러 서버의 엔터프라이즈 빈 호출

4.5.5 엔터프라이즈 빈에 의한 트랜잭션 관리

1) 개요

엔터프라이즈 빈 클래스내에 개발자가 직접 트랜잭션 관리 코드를 코딩하는 경우를 의미한다.

서버의 세션빈 소스코드 내에서의 트랜잭션 관리 코드를 빈 개발자가 직접 코딩할 수 있다. 하지만 엔티티 빈은 항상 EJB 컨테이너가 자동으로 트랜잭션을 관리해 주기 때문에 트랜잭션 코드를 삽입할 필요가 없다. 즉 엔티티 빈은 "컨테이너에 의한 트랜잭션 관리"가 기본적으로 수행된다. 반면 세션빈은 "엔터프라이즈 빈에 의한 트랜잭션 관리"와 "컨테이너에 의한 트랜잭션 관리"중 하나가 가능하다. (둘 다 동시는 불가능하다.)

트랜잭션 관리 코드는 JDBC를 사용하는 경우와 JTA를 사용하는 경우로써 두가지의 경우로 나눠진다. 두개가 같은 효과를 내지만 JTA를 사용하는 것이 데이터베이스에 종속되지 않고 좀 더 유연하게 트랜잭션을 관리할 수 있기 때문에 더 낫다.

JDBC를 사용하는 경우는 DBMS시스템의 트랜잭션 기능을 직접 사용하는 것이다. 아래 예처럼 ship이라는 메쏘드를 가진 어떤 엔터프라이즈 빈이 있을 때 이 ship()이 트랜잭션 하에서 동작하도록 JDBC의 트랜잭션 관리 코드를 사용하였다.. con은 java.sql.Connection클래스의 객체이다.

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

  try {
    con.setAutoCommit(false); // 트랜잭션 시작
    updateOrderItem(productId, orderId);
```

```
      updateInventory(productId, quantity);
      con.commit(); // 트랜잭션 커미트
    } catch (Exception ex) {
       try {
         con.rollback(); // 에러가 발생했을 경우 트랜잭션 롤백
         throw new EJBException("Transaction failed: " +
           ex.getMessage());
       } catch (SQLException sqx) {
          throw new EJBException("Rollback failed: " +
            sqx.getMessage());
       }
    }
  }
```

<예제 2> JDBC에 의한 트랜잭션 관리

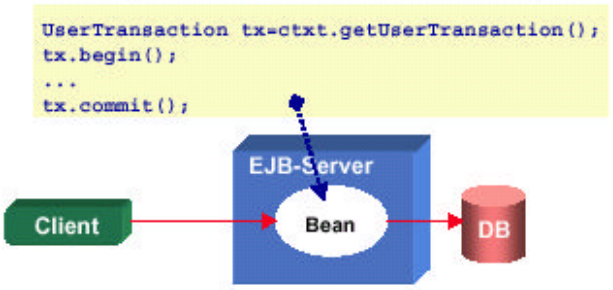
JDBC를 사용하지 않고 JTA 객체를 이용하여 트랜잭션을 관리할 수 있다.(JTA는 위의 4.3.4에서 설명하였다.) 아래 예에서 context는 SessionContext 의 상위 인터페이스인 EJBContext 인터페이스 타입의 객체이다. context는 EJB객체와 빈 인스턴스가 연관될 때 setSessionContext(SessionContext a)메쏘드를 통해서 EJB컨테이너가 자동으로 할당한다. 할당받은 SessionContext 객체의 getUserTransaction()을 통해 UserTransaction객체의 레퍼런스를 얻을 수 있다.

```
public void withdrawCash(double amount) {

UserTransaction ut = context.getUserTransaction(); // UserTransaction객체 얻음

  try {
    ut.begin(); // 트랜잭션 시작
    updateChecking(amount);
    machineBalance -= amount;
    insertMachine(machineBalance);
    ut.commit(); // 트랜잭션 커미트
  } catch (Exception ex) {
     try {
       ut.rollback(); // 에러가 발생했을 때 트랜잭션 롤백
     } catch (SystemException syex) {
        throw new EJBException
          ("Rollback failed: " + syex.getMessage());
     }
     throw new EJBException
```

```
            ("Transaction failed: " + ex.getMessage());
        }
    }
```

<예제 3> JTA에 의한 트랜잭션 관리

JTA의 트랜잭션 관리의 구조는 다음 그림과 같다. ctxt 는 SessionContext 객체이다.



<그림 12> JTA의 트랜잭션 관리

2) 상태유지 세션 빈 VS 무상태 세션빈

무상태 세션빈의 경우 트랜잭션하에 실행되는 비즈니스 메쏘드는 반드시 메쏘드의 실행이 끝나기전에 커미트하거나 롤백해야 한다. 하지만 상태유지 세션빈은 EJB 빈 메쏘드가 끝날 때 반드시 트랜잭션을 커미트하거나 롤백할 필요가 없다. 따라서 상태유지 세션빈의 경우 한 트랜잭션이 여러 개의 메쏘드에 걸칠 수 있다.

"엔터프라이즈 빈에 의한 트랜잭션 관리"에서는 EJBContext 의 getRollbackOnly()와 setRollbackOnly()를 사용해서는 안된다. 이 메쏘드는 "컨테이너에 의한 트랜잭션 관리"인 경우에만 사용될 수 있다.

아래 예는 상태유지 세션빈에서 트랜잭션이 여러 메쏘드에 걸쳐서 있는 경우이다. method1()메쏘드에서 트랜잭션이 시작하여 method3()메쏘드에서 트랜잭션이 종료한다.

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

    public void method1(...) {
            java.sql.Statement stmt;
            // EJBContext객체를 통해 UserTransaction객체를 얻음
            ut = ejbContext.getUserTransaction();
            // 트랜잭션 시작
            ut.begin();
    }
```

```
    public void method2(...) {
            javax.sql.DataSource ds;
            java.sql.Connection con;
            java.sql.Statement stmt;
            // 데이터베이스 연결을 설정한다.
            ds = (javax.sql.DataSource)
    initCtx.lookup("java:comp/env/jdbcDatabase");
            con = ds.getConnection();
            // 데이터베이스에 업데이트한다.
            stmt = con.createStatement();
            stmt.executeUpdate(...);
            stmt.executeUpdate(...);
            // 연결을 닫는다.
            stmt.close();
            con.close();
    }

    public void method3(...) {
            // EJBContext객체를 통해 UserTransaction객체를 얻음
            ut = ejbContext.getUserTransaction();
            // 트랜잭션을 커미트한다.
            ut.commit();
    }
    ...
}
```

<예제 4> 상태유지 세션빈의 여러 메쏘드에 걸친 트랜잭션

반면, 무상태 세션빈은 EJB 빈 메쏘드가 끝날 때 반드시 트랜잭션을 커미트해야 한다. 아래 예처럼 한 메쏘드내(someMethod()) 의 특정 지점에서 트랜잭션을 시작하였다면 반드시 메쏘드가 끝나기 전에 커미트해야 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();

            ds1 = (javax.sql.DataSource)
        initCtx.lookup("java:comp/env/jdbcDatabase1");
            con1 = ds1.getConnection();
            stmt1 = con1.createStatement();

            ds2 = (javax.sql.DataSource)
        initCtx.lookup("java:comp/env/jdbcDatabase2");
            con2 = ds2.getConnection();
            stmt2 = con2.createStatement();

            // UserTransaction객체를 얻는다.
            ut = ejbContext.getUserTransaction();

            // 트랜잭션의 시작
            ut.begin();

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

            // 트랜잭션을 커미트한다.
            ut.commit();

            // 연결을 해제한다.
            stmt1.close();
            stmt2.close();
            con1.close();
            con2.close();
    }
```

<예제 5> 무상태 세션빈의 메쏘드가 끝날 때 커미트 시도

4.5.6 컨테이너에 의한 트랜잭션 관리

1) 개요

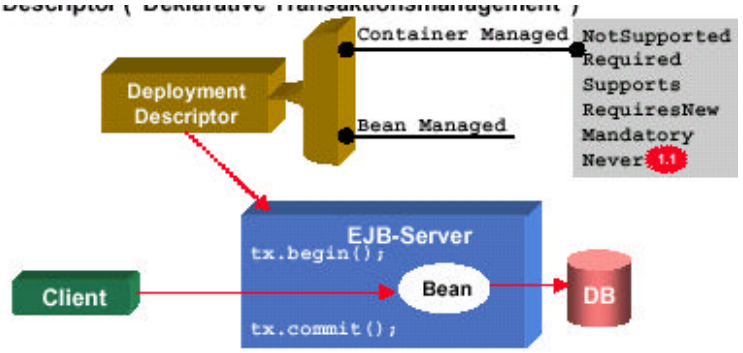
대부분의 EJB개발자가 선호하는 트랜잭션 관리 방법이다. 빈 개발자가 트랜잭션 관리 코드를 소스코드내에 포함하지 않는다. 빈 개발자가 아닌 디플로이먼트를 하는 사람이 디플로이먼트할 때 트랜잭션 속성을 지정한다. 엔터프라이즈 빈의 특정 메쏘드들, 각각이 6개의 트랜잭션 속성 중 하나를 만족하도록 디플로이먼트 디스크립터에 지정하여 디플로이먼트한다. 그러면 EJB컨테이너가 특정 메쏘드가 호출될 때 해당 트랜잭션 속성대로 트랜잭션이 보장되도록 자동으로 처리한다.

자동으로 지원되는 트랜잭션 속성은 총 6개(NotSupported, Required, Supports, RequiresNew, Mandatory, Never)가 있으며 엔터프라이즈 빈의 각 메쏘드마다 그 중 하나를 지정할 수 있다.

그럼 어떻게 EJB컨테이너가 트랜잭션을 자동화하는가? 빈개발자가 작성한 빈 클래스에는 트랜잭션 관리 코드가 없다. 그러면 그것은 어디 있는가? 답은 EJB객체이다. 2장에서 살펴본 것처럼 디플로이먼트할 때 디플로이먼트 디스크립터의 각종 정보(트랜잭션, 보안, 자원관리 등)에 따라 리모트 인터페이스를 구현(implement)한 클래스가 자동으로 생성된다. 그리고 이 클래스로부터 생성된 객체가 EJB객체이다. 리모트 인터페이스를 구현(implement)한 클래스안에 디플로이먼트 디스크립터에 지정한 트랜잭션 타입에 대응되는 트랜잭션 코드가 자동으로 생성되는 것이다. 따라서 이 클래스로부터 생성된 EJB객체가 트랜잭션 처리를 주관하는 것이다. 클라이언트는 빈 인스턴스의 존재를 모른다. 반드시 EJB객체의 비즈니스 메쏘드를 호출하고 이 메쏘드가 빈 인스턴스의 비즈니스 메쏘드를 호출한다. 트랜잭션이 자동화되었다는 것은 바로 EJB객체의 비즈니스메쏘드안에 EJB컨테이너가 삽입한 트랜잭션 코드(트랜잭션의 시작, 커미트, 롤백)가 동작하기 때문이다. 아래 그림에서처럼 클라이언트가 EJB객체의 비즈니스 메쏘드를 호출하면 먼저 트랜잭션이 시작된다. 그후 빈 인스턴스의 해당 비즈니스 메쏘드가 호출되며 이 메쏘드의 실행이 끝난후 커미트를 실행한다. 따라서 빈 인스턴스의 비즈니스 메쏘드가 트랜잭션이 보장되는 것이다.

아래 그림에서 클라이언트는 빈인스턴스의 비즈니스 메쏘드들을 호출할 수 있는데 사실 빈인스턴스 앞에는 EJB객체가 있고 이 객체가 트랜잭션의 시작과 커미트를 실행한다

.



<그림 13> 컨테이너에 의한 트랜잭션 관리의 구조

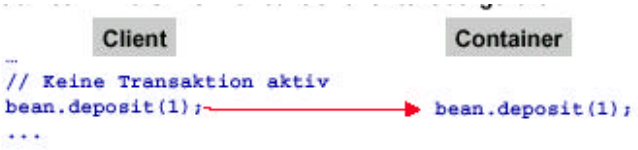
다음과 같은 메쏘드들은 컨테이너가 관리하는 트랜잭션을 방해하기 때문에 사용할 수 없다.

- java.sql.Connection의 commit, setAutoCommit, rollback 메쏘드
- javax.ejb.EJBContext의 getUserTransaction 메쏘드
- javax.transaction.UserTransaction의 모든 메쏘드

"컨테이너에 의한 트랜잭션 관리"의 경우 디플로이먼트할 때 엔터프라이즈 빈 메쏘드마다 다음과 같은 트랜잭션 속성을 지정할 수 있다.

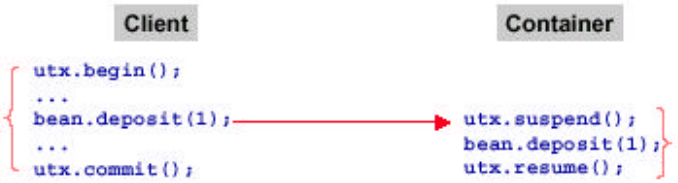
가. Not Supported

EJB 컨테이너는 명시적으로 트랜잭션을 지원하지 않으며 자동으로 트랜잭션을 결코 실행하지 않는다.



<그림 14> 클라이언트에 트랜잭션 코드가 없는 경우

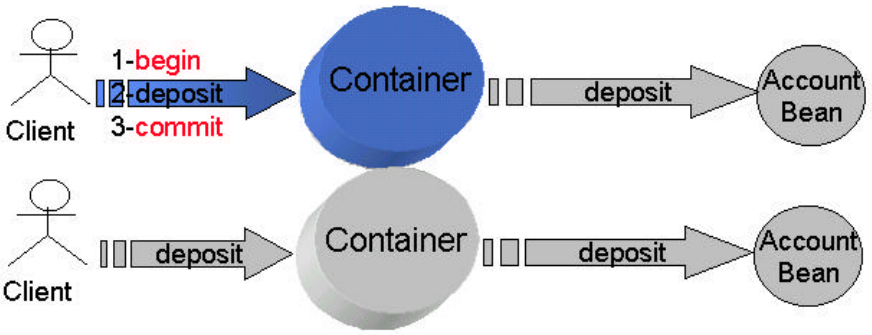
또한 클라이언트안에서 트랜잭션을 시도해도 그것을 "안 지원(Not Supported)" 한다. 만약 클라이언트 코드내에서 트랜잭션을 시도하면 엔터프라이즈 빈 메쏘드 호출이 끝날 때까지 클라이언트 트랜잭션은 멈춘 상태(suspend)가 된다.



<그림 15> 클라이언트내에 트랜잭션 코드가 있는 경우

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

아래 그림은 클라이언트가 트랜잭션하에서 deposit()라는 엔터프라이즈 빈의 메쏘드를 호출했으나 비즈니스 메쏘드는 트랜잭션 하에서 동작하지 않음을 보여준다.(푸른색은 트랜잭션하에서 동작함을 의미한다.)



]

<그림 16> "Not Supported" 트랜잭션 속성의 실행예

나. Supports

컨테이너는 자동으로 트랜잭션을 결코 실행하지 않는다.

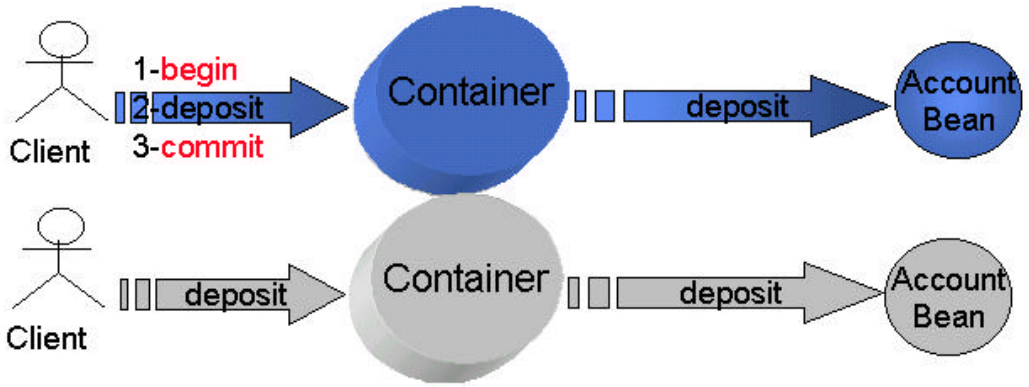
<그림 17> 클라이언트에 트랜잭션 코드가 없는 경우

그렇지만 클라이언트내에서 트랜잭션을 시도하면 그것을 "지원(Support)"한다.

<그림 18> 클라이언트에 트랜잭션 코드가 있는 경우

즉, 클라이언트 코드가 트랜잭션 하에서 엔터프라이즈 빈을 호출하면 "Required"의 경우와 같이 동작한다. 클라이언트 코드가 트랜잭션이 아닌 상태에서 엔터프라이즈 빈을 호출하면 "Not Support"와 경우와 같이 동작한다.

아래 그림은 클라이언트가 트랜잭션하에서 deposit()라는 엔터프라이즈 빈의 메쏘드를 호출하면 비즈니스 메쏘드는 같은 트랜잭션하에서 동작함을 보여준다.



<그림 19> "Supports" 트랜잭션 속성

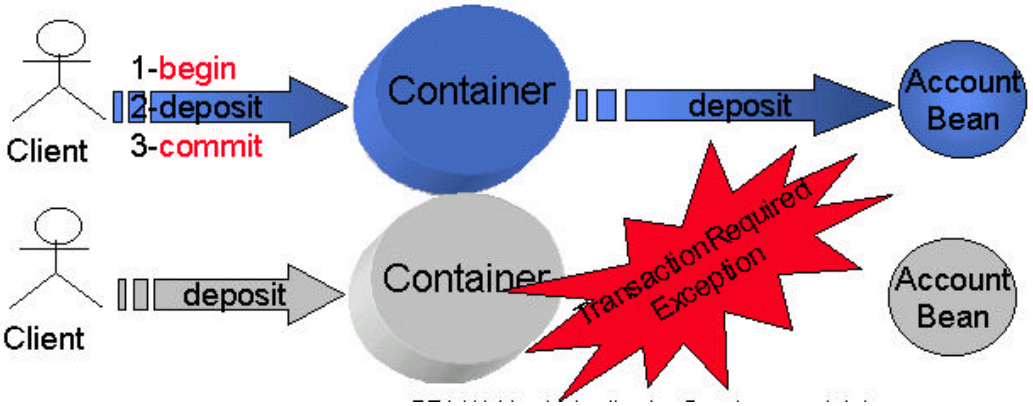
"Supports"속성은 엔터프라이즈 빈 메쏘드가 반드시 트랜잭션하에서 동작할 필요가 없는 경우에 선언한다.

다. Mandatory

클라이언트의 엔터프라이즈 빈의 메쏘드 호출은 반드시 트랜잭션 하에서 이루어져야 한다. 컨테이너는 자동으로 트랜잭션을 결코 실행하지 않는다. 트랜잭션은 클라이언트에 의해 관리되어져야 한다.

<그림 20> 클라이언트에 트랜잭션 코드가 있는 경우

즉, 클라이언트는 트랜잭션 하에서 엔터프라이즈 빈 메쏘드를 호출하도록 "강요(Mandatory)" 받는다. 그렇지 않으면 예외가 발생되기 때문이다.
만약 클라이언트가 트랜잭션 하에서 엔터프라이즈빈의 메쏘드를 호출하지 않았을 경우에는 아래 그림처럼 예외(TransactionRequiredException)가 발생된다.



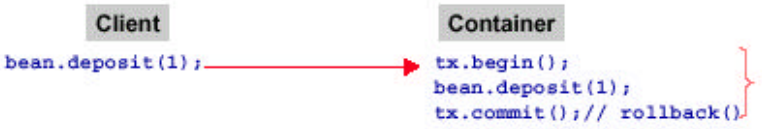
<그림 21> "Mandatory" 트랜잭션 속성

"Mandatory"속성은 엔터프라이즈 빈 메쏘드를 호출하는 클라이언트가 트랜잭션하에서 호출하지 않으면 아예 어플리케이션이 실행되지 않도록 만들 때 선언한다.

라. Required

가장 일반적으로 사용되는 트랜잭션 속성이다. 만약 어떤 트랜잭션 속성을 지정해야 할지 잘 모른다면 이 옵션을 쓰는 것이 가장 안전하다.
엔터프라이즈 빈 메쏘드는 클라이언트내에 트랜잭션 코드가 있건 없건 간에 항상 트랜잭션 상태에서 실행된다.

만약 클라이언트가 트랜잭션 하에서 동작하지 않으면 EJB 컨테이너는 자동으로 트랜잭션을 시작한다.



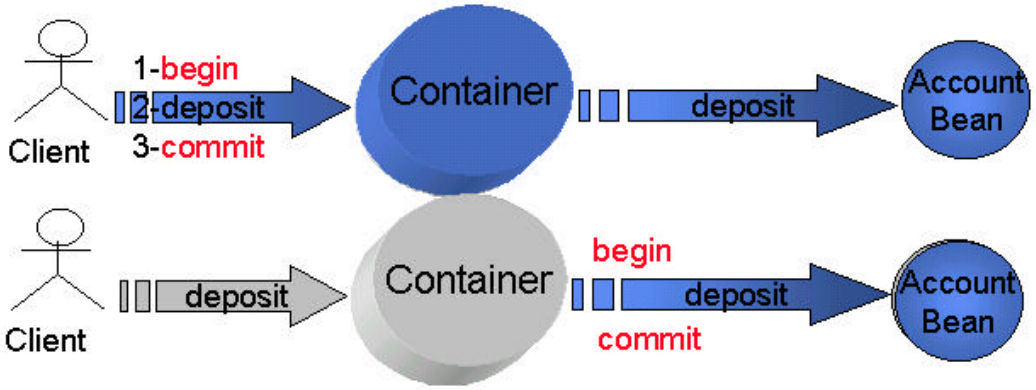
<그림 22> 클라이언트에 트랜잭션 코드가 없는 경우

만약 클라이언트내에서 트랜잭션을 시도하면 비즈니스 메쏘드도 트랜잭션하에서 동작한다.

<그림 23> 클라이언트에 트랜잭션 코드가 있는 경우

엔터프라이즈 빈 메쏘드는 컨테이너의 도움을 받아서 "필연적(Required)"으로 트랜잭션 하에서 실행된다. 만약 호출된 엔터프라이즈 빈 메쏘드가 다른 엔터프라이즈 빈 메쏘드를 호출해도 같은 트랜잭션 문맥하에서 동작한다.

아래 그림은 클라이언트가 트랜잭션을 시도하지 않아도 컨테이너가 자동으로 트랜잭션을 관리하는 것을 보여준다. 만약 클라이언트가 트랜잭션을 시도하면 그 트랜잭션을 그대로 유지한다.



<그림 24> "Required" 트랜잭션 속성

"Required"속성은 엔터프라이즈 빈 메쏘드내에서 데이터베이스나 다른 트랜잭션 리소스 관리자를 통해 데이터를 업데이트하는 경우에 사용한다. 데이터베이스의 업데이트들이 Atomicity가 됨을 완벽히 보증한다.

마. RequiresNew

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

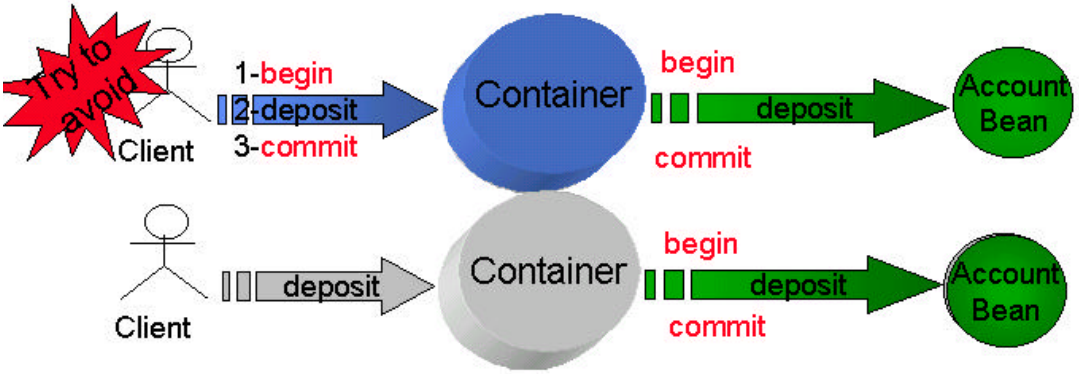
<그림 25> 클라이언트에 트랜잭션 코드가 없는 경우

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

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

<그림 26> 클라이언트에 트랜잭션 코드가 있는 경우

아래 그림에서 클라이언트가 트랜잭션을 시도하면 컨테이너는 또다른 트랜잭션을 시작하는 것을 볼 수 있다.(녹색은 또 다른 트랜잭션이다.)



<그림 27> "RequiresNew" 트랜잭션 속성

"RequiresNew"속성은 엔터프라이즈 빈 메쏘드내에서 데이터베이스나 다른 트랜잭션 리소스 관리자를 통해 데이터를 업데이트하는 경우에 사용된다는 점에서 "Required"와 비슷하다. 하지만 클라이어트의 트랜잭션과 상관없이 메쏘드내의 데이터베이스의 업데이트들이 커미트된다는 것이 다르다.

사. Never

트랜잭션이 "어떤 상황에서도 실행되지 않는다(Never)". 만약 클라이언트 코드가 트랜잭션 하에서 엔터프라이즈 빈 메쏘드를 호출하면 java..rmi.RemoteException이 발생한다. 따라서 클라이어트는 트랜잭션 하에 있으면 안된다. 만약 클라이언트 코드가 트랙잭션 상태가 아니라면 "NotSupport"경우와 같이 동작한다.

<그림 28> 클라이언트에 트랜잭션 코드가 없는 경우

"Never"속성은 엔터프라이즈 빈 메쏘드가 트랜잭션을 도저히 감당할 수 없을 때 아예 클라이언트가 트랜잭션하에서 동작하지 못하도록 만든다.

이때까지 6개의 트랜잭션들을 살펴보았으며 그것을 요약한 내용은 다음과 같다.(T1 : 클라이어트코드 내에서 시작된 트랜잭션, T2 : EJB컨테이너에 의해 시작된 트랜잭션)

트랜잭션 속성	클라이언트의 트랜잭션	엔터프라이즈 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

<표 3> 6개 트랜잭션 속성의 요약

엔터프라이즈 빈의 비즈니스 메쏘드는 다른 엔터프라이즈 빈의 비즈니스 메쏘드를 호출할 수 있다. 이때 호출되는 엔터프라이즈 빈의 비즈니스 메쏘드도 호출한 엔터프라이

즈빈에서 시작한 트랜잭션이 연결되는데 이것을 "엔터프라이즈 빈 메쏘드와 연관된 트랜잭션"이라고 한다. 자원관리자는 JDBC 드라이버와 같이 트랜잭션관련 자원을 관리하는 주체이다.

2) 트랜잭션 속성을 지정하는 구체적인 방법

가. 디플로이먼트 디스크립터안에 선언

디플로이먼트 디스크립터안에 메쏘드 단위로 6개의 EJB트랜잭션 속성중 하나를 직접 지정할 수 있다.

먼저 <enterprise-beans>…</enterprise-beans>안에 트랜잭션 타입을 Container로 선언해야 한다. 예를 들어 "<enterprise-beans>…<session>…<transaction-type>Container</transaction-type>…</session>…</enterprise-beans>"이다.

그 다음 <assembly-descriptor>…</assembly-descriptor>안의 각 메쏘드마다 트랜잭션 속성을 지정한다. 아래의 예에서처럼 <container-transaction>…</container-transaction>의 내용이 바로 메쏘드에 트랜잭션 속성을 선언한 것이다. 예를 들어 "<container-transaction><method><ejb-name>EmployeeRecord</ejb-name><method-name>updatePhoneNumber</method-name></method><trans-attribute>Mandatory</trans-attribute></container-transaction>"는 EmployeeRecord 엔터프라이즈 빈의 updatePhoneNumber라는 메쏘드가 "Mandatory"타입의 트랜잭션 속성으로 선언되었다는 뜻이다. 클라이언트가 디플로이먼트된 EmployeeRecord 엔터프라이즈 빈의 updatePhoneNumber메쏘드를 호출하면 "Mandatory"트랜잭션 속성대로 트랜잭션이 자동으로 관리되어진다.

트랜잭션 속성이 선언된 디플로이먼트 디스크립터의 예는 다음과 같다.

```
<ejb-jar>
<enterprise-beans>
<session>
    …
    <transaction-type>Container</transaction-type>
    …
</session>
…
</enterprise-beans>

<assembly-descriptor>
…
<container-transaction>
<method>
```

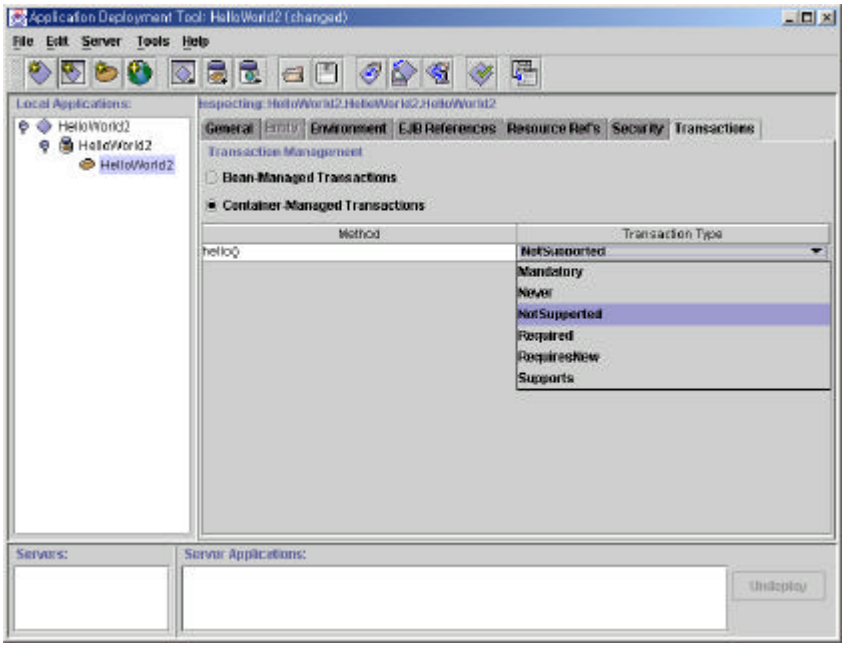
```
<ejb-name>EmployeeRecord</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>

<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>
```

<예제 6> 트랜잭션 속성이 선언된 디플로이먼트 디스크립터

나. 디플로이먼트 툴에서 GUI로 제공

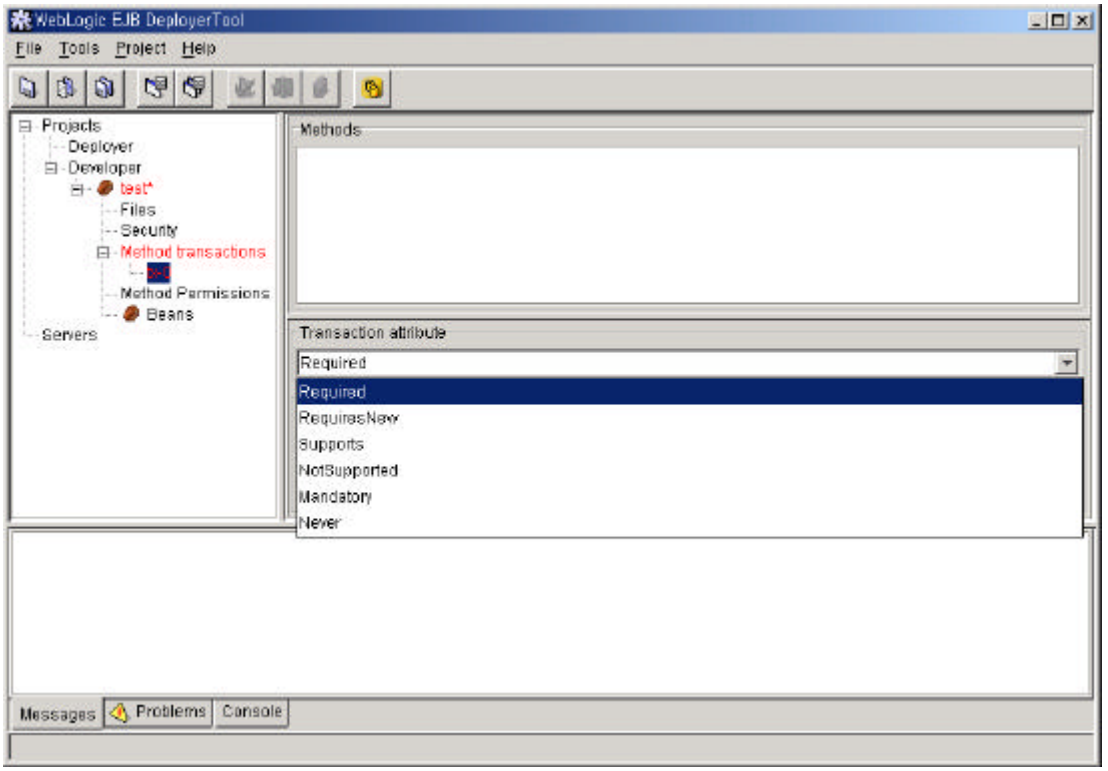
EJB컨테이너 업체에서 제공하는 디플로이먼트 툴에서 트랜잭션 속성을 비주얼하게 선언할 수 있다. 이때 위의 디플로이먼트 디스크립터가 자동으로 생성된다.

J2EE SDK의 경우는 다음 그림과 같이 메쏘드마다 트랜잭션 속성을 지정할 수 있다.



<그림 29> J2EE SDK 디플로이먼트 툴의 트랜잭션 속성 선언

웹로직의 경우는 다음과 같다.



<그림 30> 웹로직 디플로이먼트 툴의 트랜잭션 속성 선언

3) 예제 소스코드

빈 클래스를 작성할 때 개발자는 트랜잭션의 시작, 커미트, 롤백을 신경 쓸 필요없이 데이터베이스 관련 코드를 코딩할 수 있다.

간단한 세션빈의 샘플 코드는 다음과 같다. someMethod()안에서 트랜잭션을 시작, 커미트, 롤백하는 코드를 개발자가 작성할 필요가 없이 자동으로 관리된다.

```
public class MySessionEJB implements SessionBean {
SessionContext ejbContext;

// 디플로이먼트할 때 트랜잭션 속성이 선언될 것이다.
public void someMethod(...) {
java.sql.Connection con1;
java.sql.Connection con2;
java.sql.Statement stmt1;
java.sql.Statement stmt2;

// 데이터베이스 연결 객체를 얻는다.
con1 = ...;
con2 = ...;
stmt1 = con1.createStatement();
stmt2 = con2.createStatement();

/*
데이터베이스의 데이터를 업데이트한다. 이때 트랜잭션을 컨테이너가 자동으로 관리한다.
*/

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

// 연결을 닫는다.
con1.close();
con2.close();
```

```
}
…
}
```

<예제 7> 샘플 세션 빈 코드

SessionContext 객체를 통해 javax.ejb.EJBContext의 setRollbackOnly() 메쏘드를 호출할 수 있다. 이것은 트랜잭션이 결코 커미트될 수 없도록 해주는 메쏘드이다. 트랜잭션하에서 실행되는 메쏘드가 중간에 어플리케이션 관련 예외가 발생할 경우 자동으로 롤백되지 않는다. 따라서 컨테이너에게 롤백시켜달라고 알리는 메쏘드이다. 이 메쏘드를 빈 클래스내에서 호출하면 적당한 시기에 컨테이너에 의해 트랜잭션은 강제로 롤백된다. 따라서 매우 중요한 메쏘드이다.

또한 javax.ejb.EJBContext의 getRollbackOnly() 메쏘드는 현재 트랜잭션이 setRollbackOnly()에 의해 롤백되도록 셋업되었는지 확인하는 메쏘드이다.

4) 상태유지 세션빈에서 SessionSynchronization을 사용하는 경우

상태유지 세션빈은 트랜잭션이 시작, 커미트, 롤백될 때 빈 인스턴스의 속성값과 데이터베이스 정보의 일관성을 유지하기 위해 SessionSynchronization인터페이스를 구현할 수도 있다. 반드시 구현해야 하는 것은 아니다.

빈 클래스를 개발하는 개발자는 트랜잭션이 언제 시작되고 커미트되고 롤백되는지를 정확히 예측할 수 없다. 하지만 그런 시기를 알아야만 하는 경우가 있는데 그때를 위해 필요한 것이다. EJB컨테이너가 트랜잭션의 시작, 커미트, 롤백 등이 발생될 때 빈 인스턴스의 SessionSynchronization의 메쏘드들을 호출하는 것으로 그것을 구현한다. 따라서 빈 인스턴스는 현재 트랜잭션의 진행상황을 알 수 있게 되어 트랜잭션의 실행에 맞춰 적절한 행동을 할 수 있다..

가. SessionSynchronization 인터페이스 구현

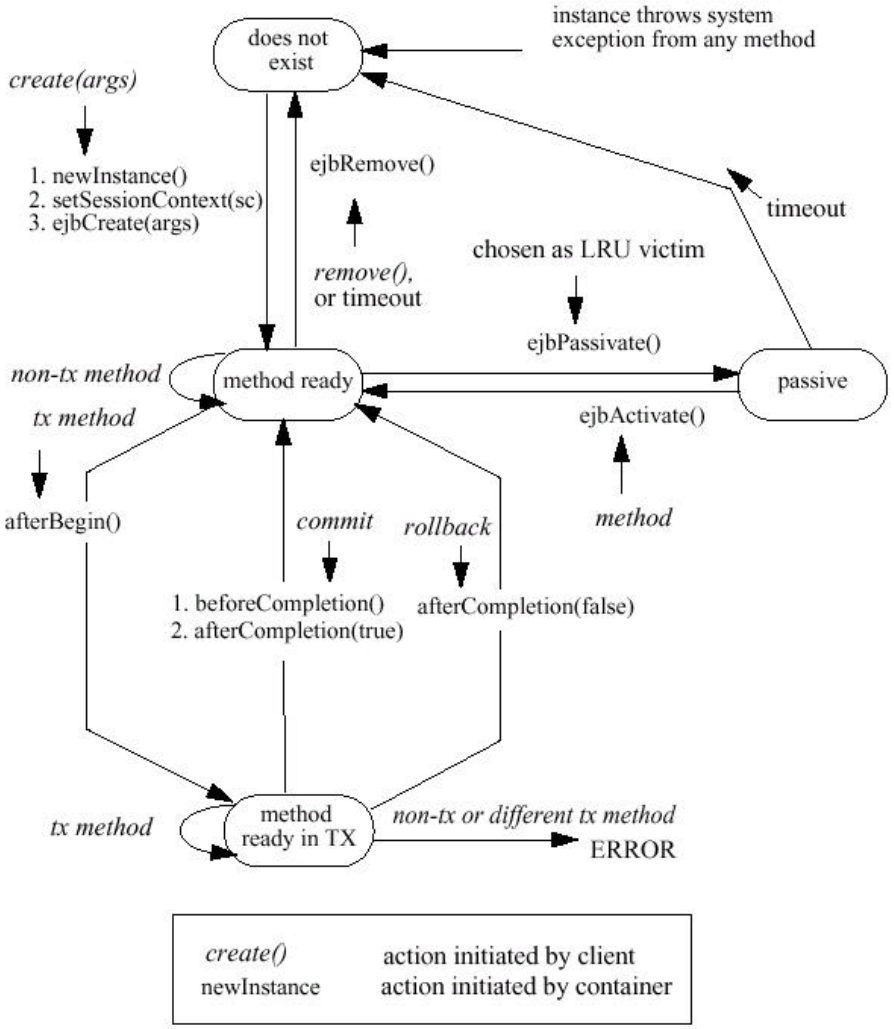
빈 클래스에서 "… implement javax.ejb.SessionSynchronization"을 통해 인터페이스를 구현한다. 이 인터페이스는 다음과 같은 3개의 메쏘드들이 선언되어있다. 즉 빈 클래스에서 이 3개의 메쏘드를 구현해야 한다.

메쏘드	기능
public void afterBegin()	트랜잭션이 시작한 직후에 컨테이너에 의해 호출됨..
public void beforeCompletion()	트랜잭션이 끝나기 직전에 컨테이너에 의해 호출됨.
public void afterCompletion(boolean committed)	트랜잭션이 끝난 직후에 호출됨. 패러미터가

	참이면 커미트, 거짓이면 롤백인 경우에 호출됨.

<표 4> javax.ejb.SessionSyncrhonization의 메쏘드

이 메쏘드들의 의미를 알기위해 상태유지 세션빈의 생명주기를 살펴보자.



<그림 31> 상태유지 세션빈의 트랜잭션 관련 생명주기

SessionSynchronization 인터페이스를 구현한 세션 빈 클래스가 있다고 가정하자. 트랜잭션 속성이 선언된 빈 인스턴스의 비즈니스 메쏘드가 호출되면 트랜잭션이 시작하기 전에 트랜잭션이 시작할 것임을 컨테이너에게 알리는 afterBegin()이 호출된다. afterBegin()이 호출되면 이후에 호출되는 비즈니스 메쏘드는 전부 트랜잭션하에서 동작하는 것이다. 따라서 이 상태에서 트랜잭션이 아니거나 다른 속성의 메쏘드가 호출되면 에러가 발생된다. 트랜잭션이 종료할 때쯤 커미트를 시도하기 직전에 beforeCompletion()이 호

출되어 빈 인스턴스에게 트랜잭션이 커미트될 것임을 알린다. 그 다음 트랜잭션이 정상적으로 커미트되면 afterCompletion(true)가 호출된다. 만약 비정상적으로 종료되어 롤백이 발생되면 beforeCompletion()이 호출되지 않고 afterCompletion(false)가 호출된다.

나. 왜 필요한가?

그럼 SessionSyncrhonization이 왜 필요한 것인가? 반드시 구현할 필요도 없는데 말이다. 아래 소스를 살펴보면서 그 이유를 찾아보자. Count 세션빈은 현재 보유하고 있는 숫자값을 1만큼 증가하는 서비스를 제공한다. Count 세션빈의 count() 메쏘드는 디플로이먼트 디스크립터에 "Required" 트랜잭션 속성이 지정되어 있다고 가정한다.

클라이언트가 count() 메쏘드를 호출하면 이것은 EJB컨테이너의 트랜잭션하에서 실행된다. 위의 생명주기를 보자. 먼저 count()메쏘드가 호출되면 count()메쏘드의 내부가 실행되기 전에 트랜잭션이 시직할 것임을 알리는 afterBegin()이 호출된다. 그 다음에 count()메쏘드의 내부가 호출된다. 즉 "return ++val;"이 호출된다. 트랜잭션 메쏘드인 count() 메쏘드가 실행이 끝난 뒤에 만약 이 메쏘드가 정상적으로 실행되었다면, 즉 커미트되었다면 beforeCompletion()이 호출되고 afterCompletion(true)가 호출된다. 패러미터가 참일 경우 아래의 메쏘드를 보다시피 "if (b==false) .."이기 때문에 아무 일도 하지 않는다. 그렇지만 count() 메쏘드가 정상적으로 실행이 되지 않았다면, 즉 롤백되었다면 afterCompletion(false)가 호출된다. 이때 현재 val값을 1만큼 감소하게 된다. 그 이유는 count()메쏘드가 이미 val값을 1만큼 증가했으나 트랜잭션이 실패하여 다시 원래 상태로 돌려 놓아야 하기 때문이다. 따라서 afterCompletion(false)를 통해 1만큼 감소해야 한다.
이렇듯 트랜잭션이 롤백되었을때 데이터베이스의 값을 돌려 놓는 것외에 빈 인스턴스의 속성도 원래 상태로 돌려야 하는 경우에 SessionSynchronization이 사용될 수 있는 것이다.

```
public class CountBean implements SessionBean, SessionSynchronization {

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 ejbPassivate() {}
public void setSessionContext(SessionContext ctx) {}
}
```

<예제 8> CountBean.java

4) 컨테이너가 관리하는 트랜잭션을 롤백하는 방법

트랜잭션이 실패하였을 경우 롤백이 제대로 수행되도록 빈 개발자는 신경써서 빈 클래스를 작성해야 한다.

"컨테이너에 의한 트랜잭션 관리"에서 롤백이 일어나야만 하는 경우는 트랜잭션 수행도중 예외가 발생하는 때이다. 이때 두가지 처리방식이 있다. 첫번째로 시스템 레벨의 예외가 발생하면 컨테이너는 자동으로 트랜잭션을 롤백한다. 두번째로는 SessionContext객체의 setRollbackOnly메쏘드를 호출하면 컨테이너가 트랜잭션을 롤백한다. 엔터프라이즈 빈내에서 시스템 레벨의 예외가 발생하면 개발자는 EJBException을 발생시키기만 하면된다. 반면 어플리케이션 레벨의 예외가 발생하면 개발자가 소스코드내에 롤백을 지시하는 코드를 직접 작성해야 한다. 왜냐하면 어플리케이션 레벨의 예외가 발생되면 롤백이 자동적으로 수행 되지 않기 때문이다. 반드시 수동으로 setRollbackOnly메쏘드를 호출함으로 롤백을 해야한다.

엔터프라이즈 빈에서 발생하는 모든 예외는 시스템 레벨의 예외와 어플리케이션 예외의 두 종류로 나눠진다. 첫번째로 시스템 레벨의 예외는 어플리케이션을 지원하는 기반 서비스에서 발생한 예외를 의미한다. 예를 들어 데이터베이스 연결이 안되는 것, 데이터베이스의 테이블이 꽉차서 삽입이 실패하는 것 등이 그것이다. 이런 예외가 발생하면 엔터프라이즈 빈은 javax.ejb.EJBException을 발생시켜야 한다. 두번째로 어플리케이션 레벨의 예외는 엔터프라이즈 빈의 비즈니스 로직에서 발생하는 예외를 의미한다. 예를 들어 사용자가 정의한 예외가 발생하거나 EJB컨테이너 계약 메쏘드에서 발생되는 CreateException, ObjectNotFoundException, RemoveException등의 예외가 그것이다. 즉 ejbCreate()에 입력 패러미터를 정확히 넣지 않았을 때 발생하는 CreateException처럼 메쏘드를 의도된 방식대로 사용하지 못했을 때 발생되는 예외도 어플리케이션 예외가 된다.

두 예외가 발생했을 경우에 롤백을 지시하는 것은 개발자의 역할이다. 아래의 예제처럼 "이체"라는 트랜잭션을 수행할 때 만약 현재 통장의 금액이 이체할 금액보다 적으면 어플리케이션 레벨의 예외인 InsufficientBalanceException을 발생시키고 진행되던 트랜잭션이 취소되어 롤백되어야 한다. 즉 데이터베이스에 쓰여진 현재 금액 값을 원상태로 돌려야 한다. 이때 SessioContext객체의 setRollbackOnly메쏘드를 호출한다. (context는 SessionContext 객체이다.) 또한 데이터베이스 관련 예외, 즉 시스템 레벨의 예외가 발생하면

EJBExeption을 발생시켜서 컨테이너가 롤백하도록 알려야 한다.

```
public void transferToSaving(double amount) throws
  InsufficientBalanceException  {

  checkingBalance -= amount; // 현재 예금계좌를 이체금액만큼 감소한다.
  savingBalance += amount; // 이체할 예금계좌를 이체금액만큼 증가한다.

  try {
    updateChecking(checkingBalance);
                //데이터베이스에 현재 금액을 반영한다.
    if (checkingBalance < 0.00) { // 만약 잔고가 0보다 작으면
      context.setRollbackOnly(); //이체할 금액이 부족한것이므로 롤백함
      throw new InsufficientBalanceException(); //어플리케이션 예외 발생
    }
    updateSaving(savingBalance); // 잔고가 0보다 크면 정상처리함.
  } catch (SQLException ex) { // 만약 SQL예외, 즉 시스템 예외가 발생하면
     throw new EJBException
              // EJBException발생하여 컨테이너가 자동롤백함.
       ("Transaction failed due to SQLException: "
       + ex.getMessage());
  }
}
```

<예제 9> 예외가 발생했을 때 "이체"트랜잭션의 롤백 테크닉

일반적으로 트랜잭션 롤백을 위한 코드 템플릿은 다음과 같다.

```
try {
  // 트랜잭션의 내용
  if (어플리케이션 예외 == true ) {
          context.setRollbackOnly(); // 컨테이너에게 롤백하도록 알림
          throw new 어플리케이션예외();
  }
  // 트랜잭션의 내용
} catch(시스템예외 ) {
  throw new EJBException(); // 컨테이너에게 롤백하도록 알림
}
```

<예제 10> 트랜잭션 롤백 코드 템플릿

4.5.7 트랜잭션 관련 통합 예제

4.5.7.1 컨테이너에 의한 트랜잭션 관리의 예제

은행업무(이체, 잔액확인)를 상징하는 Bank라는 세션빈이 컨테이너에 의해 트랜잭션이 자동으로 관리된다고 가정하자. 고객의 현재 계좌와 이체할 계좌의 잔액이 데이터베이스에 저장되어 있다.

트랜잭션은 이체를 상징하는 transferToSaving 메쏘드이다. BankBean 클래스의 속성값인 checkingBalance , savingBalance는 각각 현재계좌의 잔액과 이체할 계좌의 잔액을 의미한다. 이값은 데이터베이스에 반영되어야 한다.. 또한 SessionSynchronization인터페이스의 beforeCompletion(), afterCompletion()메쏘드를 구현하여 이 속성의 값들이 트랜잭션의 시작,커미트,롤백될 때마다 정확하게 데이터베이스의 값과 일치되도록 하였다.

1) 내용

이 EJB어플리케이션은 아래와 같은 소스코드들로 구성되어 있다.

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

2) 소스

가. Bank.java : 리모트 인터페이스

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

public interface Bank extends EJBObject {
    // "이체 " 트랜잭션 메쏘드
    public void transferToSaving(double amount) throws RemoteException;
   // 현재 통장의 잔고 확인
    public double getCheckingBalance() throws RemoteException;
   // 이체 통장의 잔고 확인
    public double getSavingBalance() throws RemoteException;
}
```

<소스> Bank.java

나. BankEJB.java : 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";

 // 이체 메쏘드 구현
 public void transferToSaving(double amount) throws
    InsufficientBalanceException  {
    checkingBalance -= amount;
    savingBalance += amount;

    try {
      updateChecking(checkingBalance); // 데이터베이스 업데이트
      if (checkingBalance < 0.00) {
        context.setRollbackOnly(); // 롤백 지시
        throw new InsufficientBalanceException();
      }
      updateSaving(savingBalance);
    } catch (SQLException ex) {
       throw new EJBException // 롤백 지시
         ("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;
  }

  // "이체" 트랜잭션(transferToSaving)이 시작된 후 호출된다.
  public void afterBegin() {
    System.out.println("afterBegin()");
try {
 // 현재 데이터베이스의 checkingBalance를 읽어서 속성에 할당한다.
      checkingBalance = selectChecking();
      // 현재 데이터베이스의 savingBalance값을 읽어서 속성에 할당한다.
      savingBalance = selectSaving();
    } catch (SQLException ex) {
       throw new EJBException("afterBegin Exception: " + ex.getMessage());
    }
}

  // 트랜잭션이 정상적으로 종료된 직후에 호출된다.
  public void beforeCompletion() {
    System.out.println("beforeCompletion()");
  }

// 트랜잭션이 커미트 또는 롤백으로 종료되었을 때 호출된다.
  public void afterCompletion(boolean committed) {
    System.out.println("afterCompletion: " + committed);
    if (committed == false) { // 트랜잭션이 롤백되었다면
      try {
      // 롤백된 데이터베이스의 원래값을 읽어서 속성에 할당한다.
// 이것을 하지않으면 현재 빈 클래스의 checkingBalance, savingBalance
// 속성에 저장된 값과 롤백된 데이터베이스의 checkingBalance,
// savingBalance값 이 다르게 된다.
        checkingBalance = selectChecking();
        savingBalance = selectSaving();
      } catch (SQLException ex) {
         throw new EJBException("afterCompletion SQLException: " +
           ex.getMessage());
      }
    }
  }

  public BankEJB() {}
```

```
/****************** 데이터베이스 관련 루틴 **********************/
 // 데이터베이스의 checkingBalance값을 패러미터값으로 쓴다.
  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();
  }
 // 데이터베이스의 savingBalance값을 패러미터값으로 쓴다.

  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();
  }

// 현재 데이터베이스의 checkingBalance값을 반환한다.
  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.");
    }
  }

// 현재 데이터베이스의 savingBalance값을 반환한다.
  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 : **홈인터페이스**

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

public interface BankHome extends EJBHome {

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

<소스> BankHome.java

라. InsufficientBalanceException.java : **사용자 정의 예외**

```
public class InsufficientBalanceException extends Exception {

  public InsufficientBalanceException() { } // 예금잔고가 부족

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

<소스> InsufficientBalanceException.java

마. createTable.sql : **데이터베이스 구조**

```
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;
```

바. 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");  //현재 계좌를 123으로 초기화
      duke.transferToSaving(40.00); // 40.00을 이체함.
```

```
            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

4.5.7.2 엔터프라이즈 빈에 의한 트랜잭션 관리 예제 1 (JDBC 사용)

창고관리(출하, 현재 재고 확인)를 상징하는 WareHouse라는 세션빈내에서 모든 트랜잭션이 관리된다고 가정하자. 현재 창고에 있는 물건의 종류 및 수량이 데이터베이스에 저장되어 있다.
트랜잭션은 출하를 상징하는 ship 메쏘드이다. JDBC를 사용하여 모드 트랜잭션 관리코드를 작성하였다.

1) 내용

이 EJB어플리케이션은 아래와 같은 소스코드들로 구성되어 있다.

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

2) 소스

가. Warehouse.java : 리모트 인터페이스

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

public interface Warehouse extends EJBObject {
  // 창고의 물건을 꺼낸다. 출하
   public void ship(String productId, String orderId, int quantity)
       throws RemoteException;
  // 재고를 확인한다.
```

```
    public String getStatus(String productId, String orderId)
      throws RemoteException;
}
```

<소스> Warehouse.java

나. WarehouseEJB.java : 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); // 자동 커미트가 안되게 셋팅한다.
      updateOrderItem(productId, orderId);
      updateInventory(productId, quantity);
      con.commit(); // 트랜잭션이 끝났으므로 커미트한다.
    } catch (Exception ex) {
       try {
         con.rollback(); // 트랜잭션 발생시 예외가 발생하면 롤백한다.
         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() {}

/********************** 데이터베이스 루틴 **********************/
    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

다. WarehouseHome.java : **홈 인터페이스**

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

public interface WarehouseHome extends EJBHome {

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

<소스> WareHouseHome.java

라. createTable.sql : **데이터베이스 구조**

```
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;
```

바. 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

4.5.7.3 엔터프라이즈 빈에 의한 트랜잭션 관리 예제 2 (JTA 사용)

은행의 ATM(예금인출,현재잔액확인)를 상징하는 Teller라는 세션빈내에서 모든 트랜

잭션이 관리된다고 가정하자. 현재 ATM 기계안에 들어있는 현금과 고객 예금계좌의 잔액이 데이터베이스에 저장되어 있다.
트랜잭션은 예금인출을 상징하는 withdrawCash 메쏘드이다. JDBC가 아니라 JTA를 사용하여 모드 트랜잭션 관리코드를 작성하였다.

1) 내용

이 EJB어플리케이션은 아래와 같은 소스코드들로 구성되어 있다.

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

2) 소스

가. Teller.java : 리모트 인터페이스

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

public interface Teller extends EJBObject {
  // ATM기계에서 현금 인출
  public void withdrawCash(double amount)
     throws RemoteException;

  // 예금계좌의 잔액 확인
  public double getCheckingBalance()
     throws RemoteException;
}
```

<소스> Teller.java

나. TellerEJB.java : 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; // 현재 기계안에 들어있는 현금
  private SessionContext context;
  private Connection con;
  private String dbName = "java:comp/env/jdbc/TellerDB";

  public void withdrawCash(double amount) {
    // 트랜잭션 관리 객체인 UserTransaction객체 얻음.
    UserTransaction ut = context.getUserTransaction();

    try {
      ut.begin(); // 트랜잭션 시작
      updateChecking(amount);.
       // 인출한 금액만큼 DB의 예금계좌값을 줄임
      machineBalance -= amount;
       // 인출한 금액만큼 기계의 현금을 줄임
      insertMachine(machineBalance);
       // 현재 기계의 현금값을 DB에 반영
ut.commit(); // 트랜잭션 커미트
    } catch (Exception ex) { // 트랜잭션 도중 예외가 발생하면
       try {
         ut.rollback(); // 트랜잭션 롤백
       } 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() {}

/************************데이터베이스 루틴 **********************/

  private void makeConnection()
    throws NamingException, SQLException {

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

// 감소된 양만큼
  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

다. TellerHome.java : **홈 인터페이스**

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

public interface TellerHome extends EJBHome {

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

<소스> TellerHome.java

라. createTable.sql : **데이터베이스의 구조**

```
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;
```

바. 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"); // 현재 기계의 현금이 123
      System.out.println("checking = " + duke.getCheckingBalance());
      duke.withdrawCash(60.00); // 60을 기계에서 인출함
      System.out.println("checking = " + duke.getCheckingBalance());
      duke.remove();
```

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

<소스> TellerClient.java

4.5.8 Isolation

4.5.8.1 개요

트랜잭션들은 작업이 "All or Nothing"으로 처리해야 할 뿐 아니라 여러 트랜잭션들이 동시에 실행될 경우 한 트랜잭션이 다른 트랜잭션에 의해 방해받지 않고 진행되어야 한다. Isolation이란 다른 트랜잭션이 여러 개의 트랜잭션이 동시에 같은 데이터를 사용해도 서로 독립적으로 해당 데이터를 처리할 수 있는 능력을 의미한다.

여러 트랜잭션이 동시에 같은 데이터를 사용하면 다음과 같은 3가지 현상이 발생할 수 있다.

a. Dirty Read

A 트랜잭션이 B 트랜잭션이 변동한 데이터를 커미트하기 전에 읽는다. 만약 B 트랜잭션이 롤백되면 A 트랜잭션은 무효한 경우가 된다.

b. Non-Repeatable Read

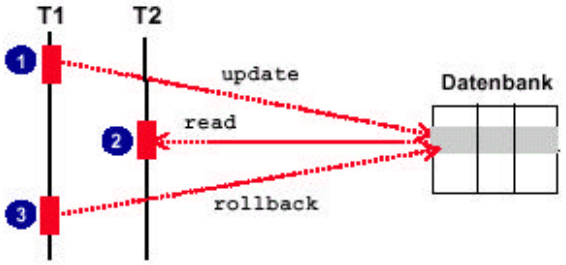
한 트랜잭션이 수행중에 읽은 데이터는 트랜잭션 중이라면 이후에 읽어도 같은 값을 가질 것을 보장할 수 없다.

c. Phantom Read

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

가. Dirty Read

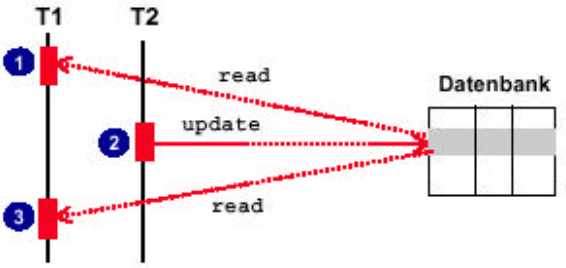
T1 트랜잭션이 T2 트랜잭션과 공유하는 테이블의 행에 값을 쓴다. 그 후 다른 트랜잭션인 T2가 T1트랜잭션이 쓴 값을 읽을 수 있다. 이때 T1이 롤백하면 테이블의 행이 T1트랜잭션 시작전의 값으로 원상복귀한다. 그러면 T2가 T1이 롤백하기 전에 읽은 데이터는 유효한 데이터가 아니다. 이런 현상을 Dirty Read라고 한다.



<그림 32> Dirty Read (Datenbank ->데이터베이스 테이블)

나. Non Repeatable Read

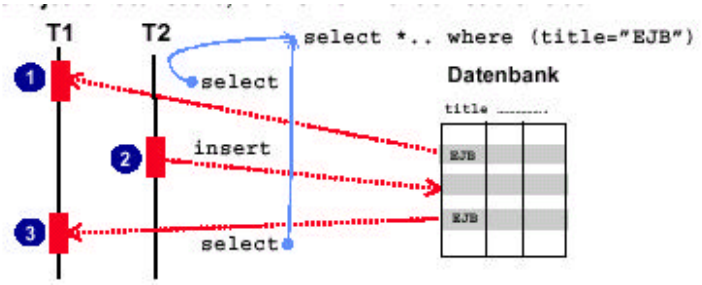
T1트랜잭션이 먼저 데이터베이스에서 데이터를 읽은 후 T2가 데이터베이스의 값을 업데이트하였따. T1이 다시 같은 데이터를 읽으면 T2가 변경한 데이터를 읽게 된다. 이런 경우를 Non-Repeatable Read라고 한다.



<그림 33> Non Repeatable Read (Datenbank->데이터베이스 테이블)

다. Phantom Read

T1트랜잭션이 "select * where (title="EJB")"와 같이 여러 행의 값을 읽은 후 T2트랜잭션이 title이 "EJB"인 행을 추가했다고 하자. select문을 만족하는 행이 추가되었음에도 불구하고 T1은 그 행의 존재를 전혀 알 수 없다. 이런 현상을 Phantom Read라고 한다.



<그림 34> Phantom Read (Datenbank->데이터베이스 테이블)

JDBC에서는 이런 현상들을 제어하기 위해 java.sql.Connection클래스의setTransaction-Isolation메쏘드로 Isolation레벨을 지정할 수 있다. 이 메쏘드에 아래와 같은 Isolation 레벨 4개 중 1개를 패러미터로 주면 아래와 같은 효과를 얻을 수 있다.

Isolation레벨	설명
TRANSACTION_READ_UNCOMMITEED	커미트되지 않은 데이터들을 읽을 수 있다.
TRANSACTION_READ_COMMITTED	커미트되지 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.

<표> Isolation 레벨 정리

Isolation 레벨과 Dirty Read, UnRepeatableRead, Phantom Read의 관계는 다음과 같다.

Isolation레벨	Dirty Reads?	UnRepeatable 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

<표> Isolation 레벨과 Dirty Read, UnRepeatble Read, Phantom Read의 관계

4.5.8.2 EJB와 Isolation

EJB 아키텍처에 Isolation을 위한 공식적인 API는 정의되어 있지 않고 따라서 리소스 매니저(Resource Manager)에 따라 그 방법이 달라진다. 즉 리소스 매니저가 데이터베이스인 경우 해당 DBMS가 지원하는 Isolation과 locking에 따라 실제 정확한 Isolation의 수행 행위가 결정된다.

컨테이너에 의해 트랜잭션이 관리될 경우 개발자는 컨테이너가 자동으로 Isolation레벨을 정의한다. 반면 엔터프라이즈 빈에 의한 트랜잭션이 관리되는 세션빈의 경우 개발자가 JDBC의 java.sql.Connection클래스의 setTransactionIsolation메쏘드를 사용하여 빈 클래스내에서 Isolation레벨을 정의할 수 있다. 이런 Isolation레벨을 엔터프라이즈 빈내에서 정의하지 않으면 데이터베이스 시스템이 디폴트로 제공하는 Isolation레벨대로 동작한다. 만약 트랜잭션의 Isolation을 개발자가 정의한다면 어플리케이션이 사용하는 DBMS가 제공하는 Isolation을 고려해야 하기 때문에 매우 주의해야한다.