Advanced Java Programming

Transactions

\( v^3 \)

Based on notes by Wayne Brooks & Monson-Haefel, R "Enterprise Java Beans 3rd ed".

Topics

- Transactions background
  - Definition, basics, properties, models
- Java and transactions
  - JTA, JTS
  - EJB support
  - Transaction scope, attributes
  - Isolation, locking
- Transactions and enterprise Java applications
  - Web transactions - Declarative, programmatic
  - EJB transactions – Declarative, programmatic

Background

- Transactions "traditionally" for databases

  Many applications have transactional requirements
  - Eg: the following 2 must complete as 1 transaction
    - deposit(acct1, 1000)
    - withdraw(acct2, -1000)

Transaction Definition

- A transaction is a “unit of work”
  - Accesses one or more shared resources
  - A set of one or more related activities
  - Activities must be completed together or not at all
  - Cohesion of unit is normally mission critical
  - Eg:
    - Bank
      - withdraw from one source, deposit to another
    - Order System
      - locate item, charge account, schedule shipment

Transaction basics

- Usually have three main operations:
  - begin transaction
  - commit transaction
  - abort transaction

Transaction properties

- All transactions have the following properties (usually called “ACID” properties)
  - Atomicity
    - “all or nothing” happens
  - Consistency
    - Different objects accessing data see a consistent view
    - System is always in a “consistent” state
  - Isolation
    - Cannot view intermediate results of other transactions
  - Durability
    - Ensure that committed data remains committed, and uncommitted data is rolled back, in the event of failure
ACID properties

- Transaction manager responsible for ensuring ACID properties are maintained
- Transaction manager might be:
  - DBMS eg: Oracle
  - TP monitor eg: IBM CICS, Bea Tuxedo
  - Java application server

- your applications don’t have to deal with all the possible things that could go wrong during a transaction
  - you leave it to the container / transaction manager

Transaction Models

- Different transaction managers support mainly 2 transaction models
  - Flat
  - Nested

- J2EE application servers only have to support the "flat" transaction model

Flat Transaction Model

![Flat Transaction Model Diagram](image)

**Figure 23.8**
The flat transaction model. From Monson-Haefel, R "Enterprise JavaBeans 3rd ed."

Nested Transaction Model

![Nested Transaction Model Diagram](image)

**Figure 23.10**
The nested transaction model. From Monson-Haefel, R "Enterprise JavaBeans 3rd ed."

Transaction Management

- Transaction manager ensures ACID properties are maintained
  - even over a network

- X/Open has a distributed transaction processing standard (based on two phase commit)

- Two phase commit (2PC)
  - Phase 1: Prepare to commit
  - Phase 2: Actual commit

Topics

- Transactions background
  - Definition, basics, properties, models
  - → Java and transactions
    - JTA, JTS
    - EJB support
    - Transaction scope, attributes
    - Isolation, locking

- Transactions and enterprise Java applications
  - Web transactions - Declarative, programmatic
  - EJB transactions – Declarative, programmatic
Java & Transactions

- Two key acronyms:
  - JTA – Java Transaction API
  - JTS – Java Transaction Service

- JTA is the API your code uses, if you want to deal with transactions programmatically

- JTS is the vendor implementation of transaction management in Java
  - JTS is a Java mapping of CORBA's Object Transaction Service (OTS)
  - sometimes you see references to JTS/OTS – this is why

Applications & Transactions

- Not all parts of an application require transactions

- Transactions are designed to enforce system integrity
  - however, they also introduce overhead
  - parts of the application that do not require transaction support may be declared/coded as such, to reduce overhead

- For now, let's consider transactions on a method-by-method basis

JTA

- JTA is one of the simplest Java APIs from a user (application developer) perspective

  ```java
  javax.transaction.UserTransaction:
  ```
  - void begin()
  - void commit()
  - void rollback()
  - void setRollbackOnly()
  - int getStatus()
  - void setTransactionTimeout(int seconds)

EJB Transaction Support

- J2EE allows us to manage transactions at the container or programmer level.

  - **Declarative**
    - transaction management controlled through deployment descriptor

  - **Programmatic**
    - direct programming calls to Java Transaction Service (JTS) API
    - transaction management code mixed with business code
      - change in transactional behavior requires change in business code

Transaction Scope

- A Unit of Work
  - consists of one or more tasks
    - Tasks == EJB methods
  - Ends when:
    - thread of execution completes normally
    - an exception is thrown
    - transaction is rolled back
  - The scope of each EJB method depends on the bean's declared transaction attributes

Transaction Attributes

- **Required**
  - uses existing transaction, or creates a new one

- **RequiresNew**
  - suspends any old transaction and creates a new one

- **NotSupported**
  - suspends any old transaction and runs with no trans.

- **Supports** (default with J2EE)
  - may run with transaction – depends on caller

- **Never**
  - refuses to run within a transaction

- **Mandatory**
  - refuses to run without a transaction
**EJB3 annotation**

Can do this at CLASS level:

```java
@TransactionAttribute(MANDATORY)
public class MyBean implements My {
    public void setPay(int staffId, float salary) {
        ...
    }
}
```

Or at METHOD level:

```java
public class MyBean implements My {
    @TransactionAttribute(MANDATORY)
    public void setPay(int staffId, float salary) {
        ...
    }
}
```

- Note: default transaction is SUPPORTS for class.

**Setting the Declarative Transaction Attribute**

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

**Required**

- (A) Joins the transaction scope if invoked as part of a transaction
- (B) Initiates its own transaction context

**RequiresNew**

- (A) (B) Initiates its own transaction scope regardless of being called within existing transaction
- Initiated transaction completes prior to returning to caller

**Not Supported**

- Transaction is suspended during the method of the Invoked EJB; resumes when method complete
- Transaction scope is not propagated to Invoked EJB or anything it invokes

**Supports**

- (A) Joins the transaction scope if invoked as part of a transaction
- (B) Can be invoked outside of a transaction
• (A) Throws a Transaction Exception if called within a transaction context
• (B) Must be invoked outside of a transaction context

Extracts from John Hopkins University, Enterprise Java course 605.784

Isolation and Database Locking

• What happens when two or more transactions attempt to access the same data?
  – Dirty Reads
    • first transaction reads uncommitted changes from a second transaction that may eventually be rolled back
  – Repeatable Reads
    • data guaranteed to be the same if read multiple times during the same transaction; implemented with locks or snapshots
  – Phantom Reads
    • first transaction sees new rows added by second transaction after beginning of the first transaction

Database Locks

• How do you prevent overlapping transactions from viewing/updating the shared data?
• Typical situations:
  – Read Lock
    • data is prevented from changing while transaction in progress
  – Write Lock
    • prevents other transactions from modifying the data
    • permits dirty reads by other transactions
  – Exclusive Write Lock
    • prevents other transactions from reading and modifying data
    • prevents dirty reads
  – Snapshots - frozen view of data
• Typically managed by the DBMS

Transaction Isolation Levels

• Solution: specify locking mechanism at the transaction level ie: Isolation Levels:
  – Read Uncommitted
    • can read data being modified by another transaction
    • allows dirty, non-repeatable, and phantom reads
  – Read Committed
    • cannot read data being modified by another transaction
    • allows non-repeatable and phantom reads; prevents dirty reads
  – Repeatable Read
    • cannot change data being read by another transaction
    • allows phantom reads; prevents dirty and non-repeatable reads
  – Serializable
    • transactions can neither read or write same data

Specifying Isolation Level

<weblogic-ejb-jar>
...
<transaction-isolation>
  <isolation-level> TRANSACTION_READ_COMMITTED </isolation-level>
</transaction-isolation>
<path>
  <ejb-name>Account</ejb-name>
  <method-name>*</method-name>
</path>
</weblogic-ejb-jar>
**Isolation Levels 1**

- **Never**
  - transactions are not supported

- **Read Uncommitted**
  - other objects can read uncommitted results – dangerous!
    - Example
      - T1 begins and updates a row
      - T2 reads the updated row
      - T1 rolls back
      - T2 has read uncommitted, ‘dirty’ data (‘dirty’ read)

**Isolation Levels 2**

- **Read Committed**
  - prevents ‘dirty’ reads but does not guarantee repeatable reads
    - what is a repeatable read?
      - should return the same value if read twice in a same transaction
    - Example
      - T1 begins and reads a row
      - T2 updates the row and commits
      - T1 reads the row again
      - Non-Repeatable Read as T1 sees different value to its first read

**Isolation Levels 3**

- **Repeatable Read**
  - prevents dirty and nonrepeatable reads
  - however, phantom reads are still possible
  - a phantom reads occurs if a row has been added or deleted and the transaction sees the side effect
    - Example
      - T1 reads using a WHERE clause
      - T2 inserts a row and commits
      - T1 reads using the same WHERE clause
      - A phantom read occurs if the second read contains the inserted row data
  - expensive to implement and does not provide true serializable transactions

**Isolation Levels 4**

- **Serializable**
  - prevents dirty, nonrepeatable and phantom reads
  - highest level, guarantees serialisability of transactions
  - lowest performance

**Programmatic settings**

- Isolation Level can be set programmatically:
  - `Connection.setTransactionIsolation()`
    - set for a given connection (only for BMP)
  - Constants are defined in `Connection` class
    - `conn.getTransactionIsolation` (Connection.TRANSACTION_READ_COMMITTED);

**Typical settings**

- **Transaction Attribute:**
  - **Required**
    - Ensures that an operation runs inside a transaction "just in case" it does something that should be protected by a transaction

- **Isolation Level:**
  - **Read Committed**
    - Lowest "safe" level
    - Setting higher levels offers more reliability, but reduced concurrency, reduced performance
    - Enables readers to proceed without waiting for an update to commit
    - in a web application most users are browsing the data
**Topics**

- Transactions background
  - Definition, basics, properties, models
- Java and transactions
  - JTA, JTS
  - EJB support
  - Transaction scope, attributes
  - Isolation, locking
- Transactions and enterprise Java applications
  - Web transactions - Declarative, programmable
  - EJB transactions - Declarative, programmable

**Enterprise Java Transactions**

- You may wish to use transactions:
  - in a servlet/JSP container (web server)
  - in an EJB container (application server)
- You may wish to use transactions:
  - declaratively (preferred)
  - programmatically (when you must)
- The following pages go through these combinations

**Declarative web transactions**

- Not possible!!!
- For web components (servlets/JSPs), if you want to use transactions, you must do it programmatically!
- J2EE goal: manage transactions by the EJB container.
  - your servlets/JSPs should only have presentation logic
  - they should only be formatting the data, not carrying out any business logic, therefore should **not** need transactions!

**Programmatic web transactions**

import javax.transaction.*;
import javax.naming.*;
...
public void doGet(...) {
    try {
        Context ctx = new InitialContext();
        UserTransaction ut = (UserTransaction)
            ctx.lookup("weblogic.transaction.UserTransaction");
        ut.begin();
        // do some transactional ops here, e.g. access DBMS
        ut.commit();
    } catch (Exception e) {
        ...
    }
}

**Programmatic web transaction (2)**

- Some restrictions:
  - transactions cannot span multiple web requests
  - if using multi-threaded servlet, the transaction must be started and completed only from the "main" thread
- You might use transactions in a servlet/JSP if:
  - you were building a small-scale application with no EJBs, and had the servlets/JSPs directly accessing multiple databases

**In EJB3**

@TransactionManagement(BEAN)
@Stateless
public class PayrollBean implements Payroll {
    @Resource UserTransaction utx;
    @PersistenceContext EntityManager payrollMgr;
    public void setPay(int empId, float salary) {
        utx.begin();
        payrollMgr.find(Employee.class, empId).setSalary(salary);
        utx.commit();
    }
}

...
Declarative EJB transactions

- You specify bean transactions on a method-by-method basis
- Entity beans can only use container-managed transactions
  → you have no choice about it
- Session beans can choose between container-managed transactions (declarative) and bean-managed transactions (programmatic)
  → as usual, container-managed transactions are preferred

Declarative EJB transactions (2)

- Even using declarative transactions, you might want to notify the transaction manager that the transaction must rollback – methods are provided on the EJB SessionContext:
  - void setRollbackOnly()
  - boolean getRollbackOnly()
- Recall that every EJB has access to its SessionContext through your implementation of:
  - void setSessionContext(SessionContext sc)

Declarative EJB transactions (3)

- In EJB 3 session beans:
  - declare transaction using
    ```
    @TransactionAttribute(TransactionAttributeType)
    (where transactionAttributeType is SUPPORTS, REQUIRED etc)
    ```

EJB 3 transactions

- Should be avoided
  - Container managed transactions are easier to use, and allow different objects to be more easily composed into larger transactions
- Only useful for fine-grain transactional behaviour
  - where you don't want an entire method to be part of a transaction, but just a few lines of code

Programmatic EJB transactions

```java
public class MySessionEJB implements SessionBean {
    SessionContext ejbCtx;
    ...
    public void myBusinessLogicMethod() {
        UserTransaction ut = ejbCtx.getUserTransaction();
        ut.begin();
        // perform transactional operations here
        ut.commit();
    }
}
```

- Uses pure JTA – easy!
**Container transaction support**

- The container automatically propagates transaction contexts
  - So if you create a transaction in a servlet, and from the servlet you call an EJB with the "Requires" attribute, the EJB will execute under the boundary of the servlet's transaction
  - If one EJB (e.g. session) makes calls on another EJB (e.g. entity), the transaction context is propagated

**Summary**

- Transactions are an important part of enterprise applications
  - fortunately EJB containers make it easy

- Container-managed (declarative) transactions are preferred to bean-managed (programmatic)
  - choose the right transaction attribute for each method
  - if you are unsure, "Required" is a safe bet

- Remember that all transaction management has performance implications.
  - Locking can bring throughput to a halt.