**J2EE Transactions Explained :**Enterprise applications use different resources to save data and send messages like a database or message queue. If we want to query these resources sequentially and to cancel the whole operation once a problem occurs, we have to put this query in a unit of work so that will be executed as a whole.

To make it clear, taking for example a customer that wants to sign up for a premium account in your site and he will wait 2 days to get his account activated once payment is received. Suppose that there is a problem in payment, the right choice is  not to register the user until you get paid. These 2 operations should be executed both or cancelled both. So here we englobe these 2 operations in a transaction.

**Transaction properties :**

These properties are known by the ACID acronym. Each letter stands for a property :

1. **Atomicity :** means that the whole operations are executed as a whole (all or nothing rule). This is what I explained in the example above.
2. **Consistency :** Data must be consistent in the beginning as well as the end of the transaction.
3. **Isolation :** When we have concurrent access to data, operations cannot access modified data in the current transaction.
4. **Durability:** Once the transaction finished, modified data must not be lost and should be the same afterward.

**Transaction types :**

* **Global transactions**

These type of transactions are needed when dealing with multiple resources (database, message queue..). In order to use global transactions, Java Transaction API (JTA) is needed and it’s available only in an application server environment

* **Local transactions**

Unlike global transactions, local transactions are used to deal with a single resource as a JDBC connection for a single database. Normally it’s easier to use than global transactions.

|  |  |
| --- | --- |
| **Enum Constant Summary** | |
| [**MANDATORY**](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#MANDATORY)            If a client invokes the enterprise bean's method while the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction context. |  |
| [**NEVER**](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#NEVER)            The client is required to call without a transaction context, otherwise an exception is thrown. |  |
| [**NOT\_SUPPORTED**](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#NOT_SUPPORTED)            The container invokes an enterprise bean method whose transaction attribute NOT\_SUPPORTED with an unspecified transaction context. |  |
| [**REQUIRED**](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#REQUIRED)            If a client invokes the enterprise bean's method while the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction context. |  |
| [**REQUIRES\_NEW**](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#REQUIRES_NEW)            The container must invoke an enterprise bean method whose transaction attribute is set to REQUIRES\_NEW with a new transaction context. |  |
| [**SUPPORTS**](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#SUPPORTS)            If the client calls with a transaction context, the container performs the same steps as described in the REQUIRED case. |  |

**Transaction propagation :**

It refers to what happens to a transaction when a method calls another method. Suppose we have method signup() calling method sendEmail().

**Required [REQUIRED]:**

* signup() holds a transaction T1 à sendEmail() uses transaction T1.
* Signup() doesn’t hold a transaction à sendEmail() creates a new transaction T1.

**Requires New [REQUIRES\_NEW] :**

* Signup() holds a transaction T1 à sendEmail() creates a new transaction T2.
* Signup() doesn’t hold a transaction à sendEmail() creates a new transaction T1.

**Supports [SUPPORTS] :**

* signup() holds a transaction T1 à sendEmail() uses transaction T1.
* Signup() doesn’t hold a transaction à sendEmail() doesn’t hold a transaction.

**Not Supports [NOT SUPPORTED]:**

* signup() holds a transaction T1 à sendEmail() doesn’t hold a transaction.
* Signup() doesn’t hold a transaction à sendEmail() doesn’t hold a transaction.

**Mandatory [MANDATORY]:**

* signup() holds a transaction T1 à sendEmail() uses transaction T1.
* Signup() doesn’t hold a transaction à Exception is thrown.

**Never [NEVER]:**

* signup() holds a transaction T1 à Exception is thrown.
* Signup() doesn’t hold a transaction à sendEmail() doesn’t hold a transaction.

**Nested :**

* signup() holds a transaction T1 à sendEmail() creates a nested transaction inside T1 like a savepoint. This is applied only to JDBC resources.
* Signup() doesn’t hold a transaction à sendEmail() creates a new transaction T1.

**Transaction Isolation levels :**

Isolation is the behavior of rendering data once accessed concurrently within a transaction.

Below are the isolation levels that differ in implementation for every resource.

* **Read Committed :** it ensures that data modifications in a transaction are not read until the transaction commits.
* **Read uncommitted :** unlike “Read Committed” level, data modifications are read during a transaction even if the transaction can be rolled back later. It’s called also “Dirty Read”.
* **Repeatable read:** it ensures that data will not be modified by another transaction so as data is consistent when reading it during different times of a transaction.
* **Serializable :** it ensures data is not inserted or updated until the transaction finishes.