



Mindtree

Welcome to possible

Enterprise Java Bean Dependency Injection and JTA

Objectives

- Define dependency injection using CDI
- Define transaction management using JTA

Dependency Injection

- Dependency Injection (DI) refers to the process of supplying an external dependency to a software component.
- DI can help make your code architecturally pure.
- It aids in design by interface as well as test-driven development by providing a consistent way to inject dependencies.
 - For example, a data access object (DAO) may depend on a database connection.
 - Instead of looking up the database connection with JNDI, you could inject it.
- One way to think about a DI framework like CDI is to think of JNDI turned inside out. Instead of an object looking up other objects that it needs to get its job done (dependencies), a DI container injects those dependent objects.
- This is the so-called Hollywood Principle, "Don't call us" (lookup objects), "we'll call you" (inject objects).

Contexts and Dependency Injection for the Java EE

- Contexts and Dependency Injection (CDI) for the Java EE platform is one of several Java EE 6 features that help to knit together the web tier and the transactional tier of the Java EE platform.
- CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with JavaServer Faces technology in web applications.
- Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate various kinds of components in a loosely coupled but type-safe way

Contexts and Dependency Injection for the Java EE

- **Field Dependency Injection**

- The easiest way to inject a CDI bean is to add the `@Inject` annotation in the property to be injected. Let's take a look at the example below.
- The `GreetingBean` has an `@Inject` annotated field, which is the `helloBean`. In this way another bean, the `HelloBean` is injected into the `GreetingBean`.

```
@Stateless
@Remote(GreetingIntf.class)
public class GreetingBean implements GreetingIntf{

    @Inject
    private HelloBean helloBean;

    @Override
    public String greeting(String name) {
        return helloBean.sayHello(name);
    }
}
```

Contexts and Dependency Injection for the Java EE

- **Constructor Dependency Injection**

- When a CDI bean is initialized, the container will use its default constructor.
- When there is another constructor annotated with the `@Inject` annotation, then the container will automatically use this constructor instead, and in this way the argument passed in the constructor will be injected in the bean

```
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.inject.Inject;

@Stateless
@Remote(GreetingIntf.class)
public class GreetingBean implements GreetingIntf {

    private final HelloBean helloBean;

    @Inject
    public GreetingBean(HelloBean helloBean) {
        this.helloBean = helloBean;
    }
}
```

Contexts and Dependency Injection for the Java EE

- Dependency Injection through the setter method

```
import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.inject.Inject;

@Stateless
@Remote(GreetingIntf.class)
public class GreetingBean implements GreetingIntf {

    private HelloBean helloBean;

    @Inject
    public void setHelloBean(HelloBean helloBean) {
        this.helloBean = helloBean;
    }

    @Override
    public String greeting(String name) {
        return helloBean.sayHello(name);
    }
}
```

Contexts and Dependency Injection for the Java EE

- The @Resource annotation can be used to inject env-entries, EJBContext, JMS destinations and connection factories, and datasources.

```
public class TestBean implements TestIntf {
```

```
    @Resource
```

```
    SessionContext ctx;
```

```
    @Resource(name="jdbc/pb", type=javax.sql.DataSource.class)
```

```
    private DataSource employeeDataSource;
```


Contexts and Dependency Injection for the Java EE

- EJB DI into the Servlet

```
public class ControllerServlet extends javax.servlet.http.HttpServlet  
    implements javax.servlet.Servlet {
```

```
@EJB
```

```
FetchCustomer fetchCustomerIntf;
```

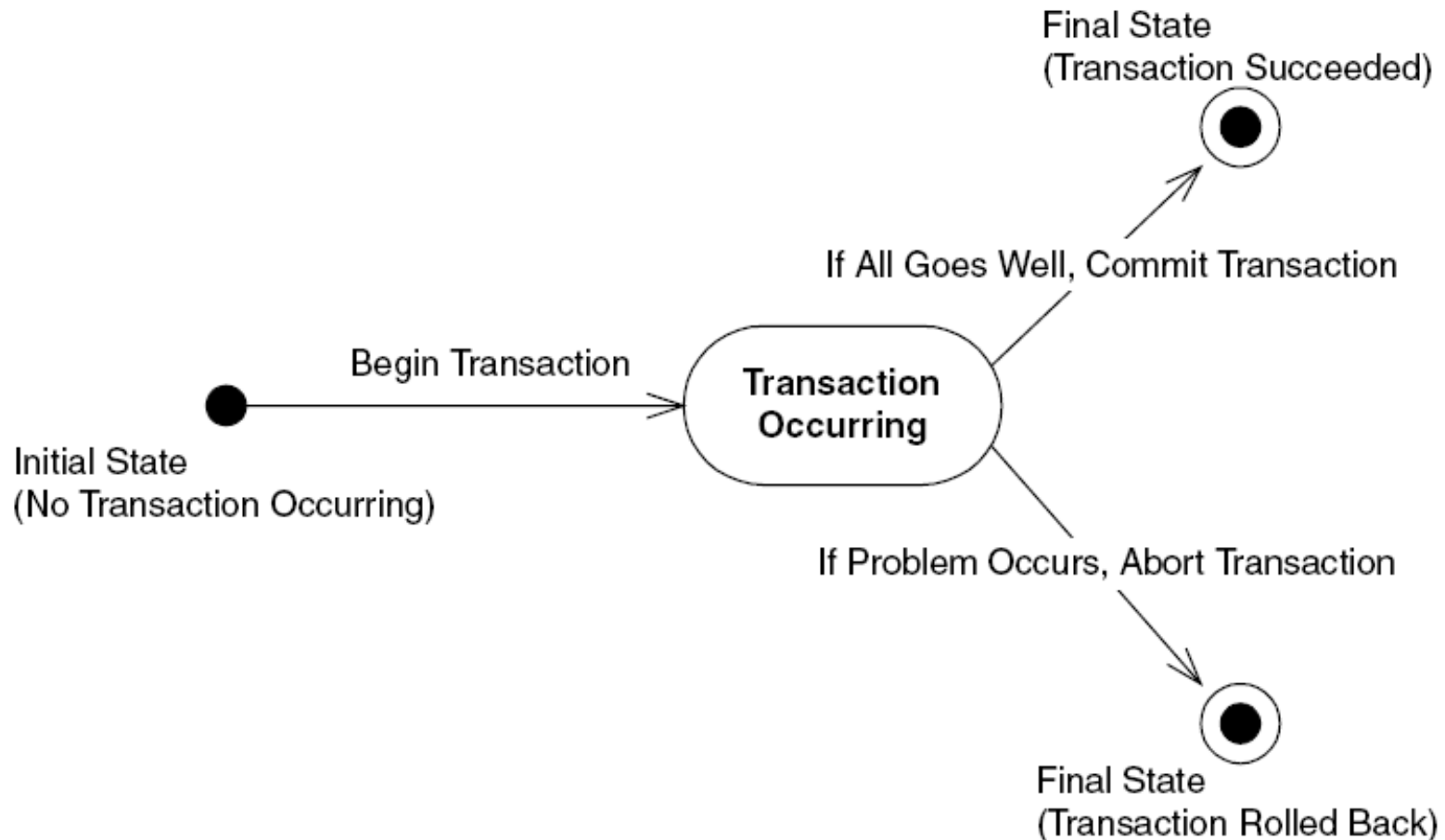
```
public void doGet(HttpServletRequest request, HttpServletResponse  
response) throws ServletException, IOException {
```

```
    List<String> names = fetchCustomerIntf.getCustomerNames();
```

Transactions

- ACID
 - Atomicity guarantess that many operations are bundled together and appear as one contiguous **unit of work**. (All or nothing paradigm)
 - Consistency guarantess that a transaction leaves the system's state to be consistent after transaction completes.
 - Isolation protects concurrently executing transactions without seeing each other's incomplete results.
 - Durability guarantess that the updates to managed resources , such as database records survive failures.

Flat Transactions

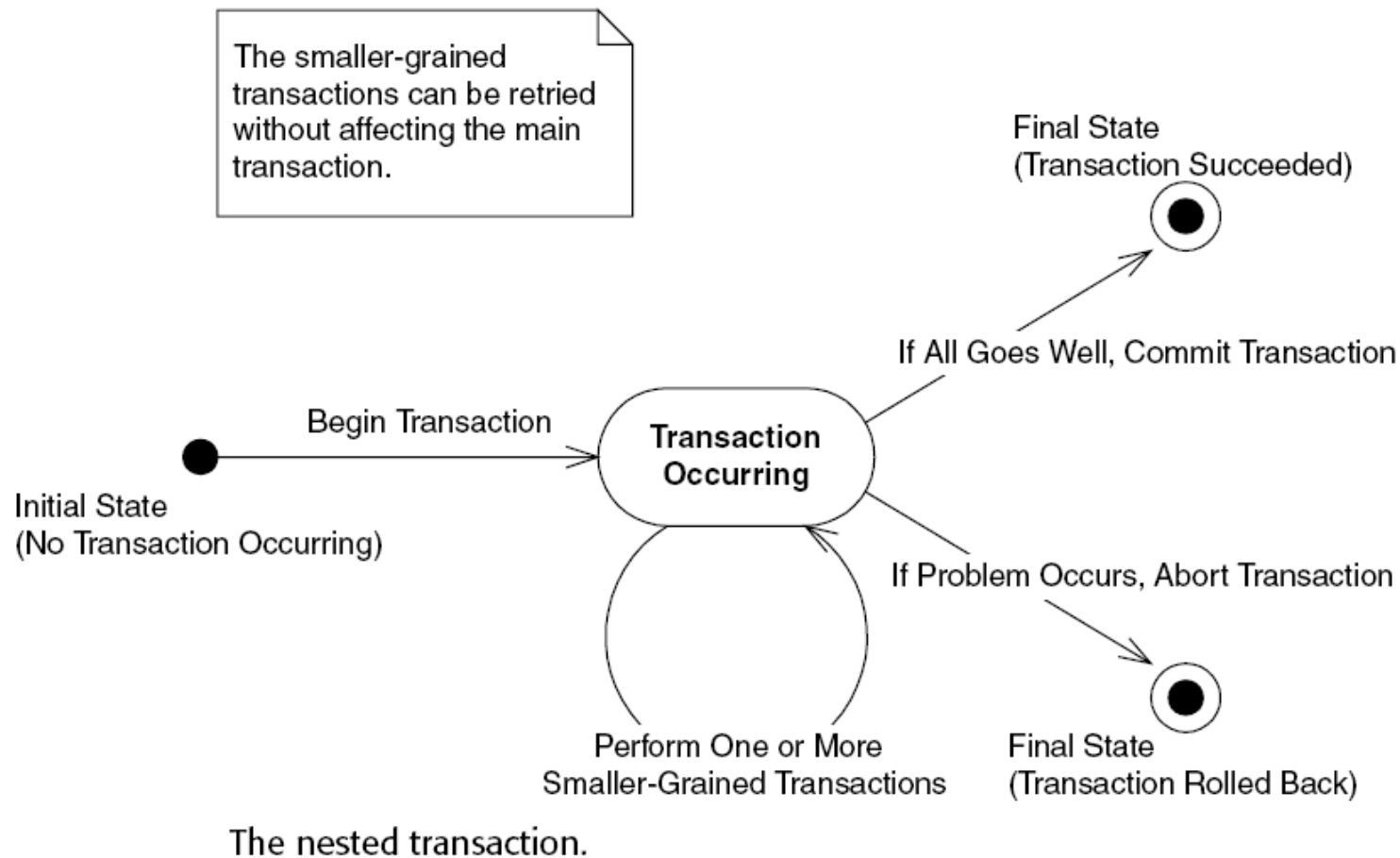


The flat transaction.

Nested Transactions

- Consider Your application performs the following operations for your trip.
- Purchase a train ticket from Mysore to Bangalore
- Purchase a Flight ticket from Bangalore to Kochi
- From Kochi you take a cruise to Maldives
- Your application finds out that there are no outgoing flights from Maldives.
- If this sequence of Bookings happen as Flat transaction, your application would have only one option to rollback the entire transaction.
- Famous trip-planning problem.
- Nested transaction enables you to embed atomic units of work within other units of work.
- Larger unit of work can attempt to retry the embedded unit of work.
- If the embedded unit of work can be made to succeed , the larger unit can be made to succeed.

Nested Transactions

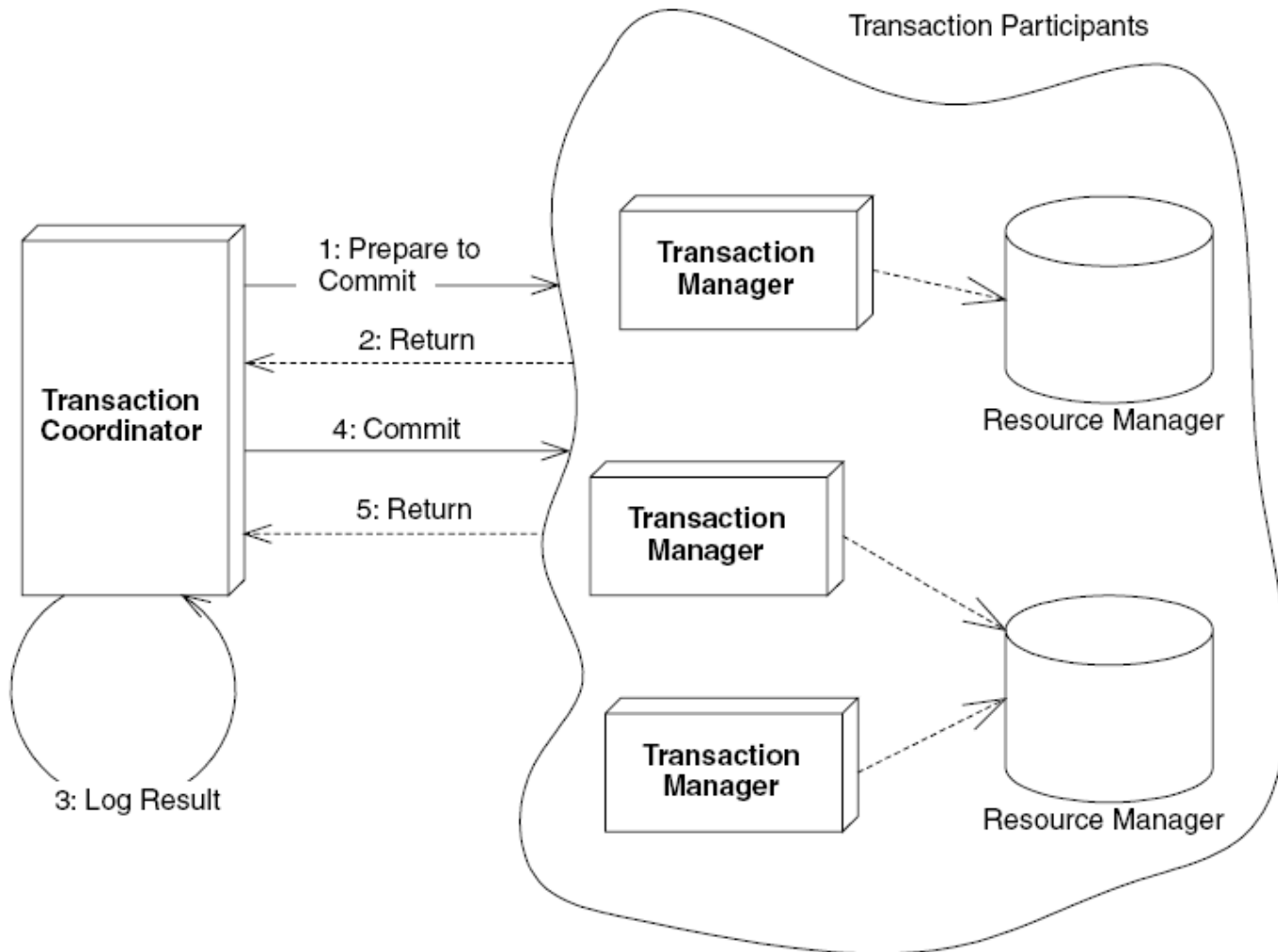


Distributed Transactions

Distributed Transactions are applicable if

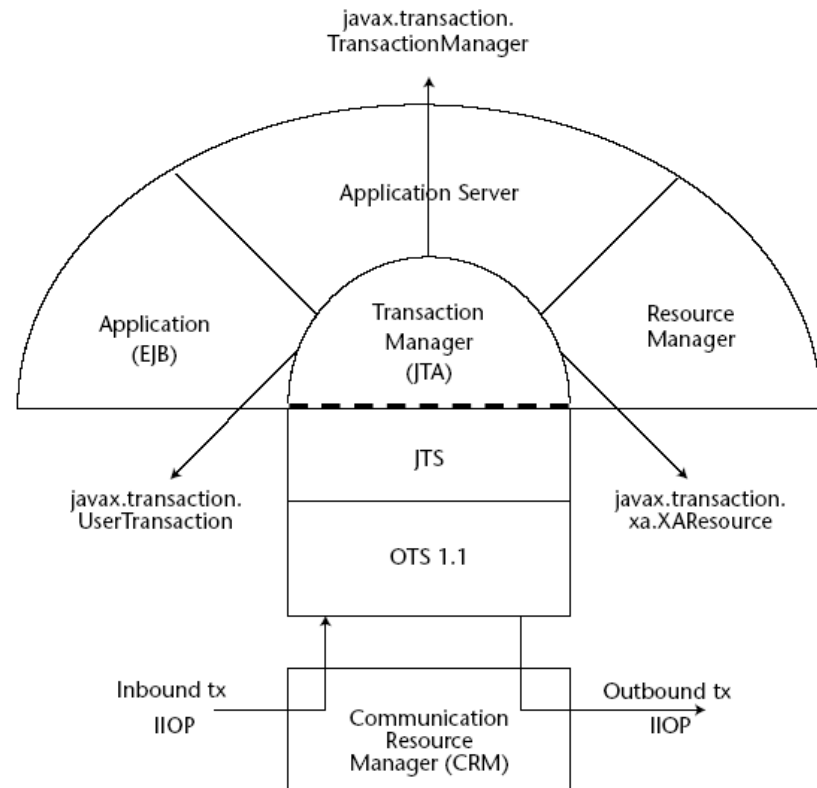
- You have multiple application servers coordinating in the same transaction
- You have to updates to different databases in the same transaction
- You are trying to update database and send and receive a JMS message
- You are connecting to a legacy system as well as one or more other types of resource (Database, message queues , or other legacy systems) in same transaction.

Distributed Transactions and Two Phase commit Protocol



Java Transaction Service(JTS) and Java Transaction API (JTA)

- CORBA's OTS (Object Transaction Service) support multiple parties participating in a transaction.
- EJBs Expert group choose to reuse lot of work that went into CORBA
- JTA defines the core of transaction supports in EJB. It defines interfaces between a transaction manager and other parties , namely enterprise beans, resource managers and application servers.
- JTA is based upon low level JTS.

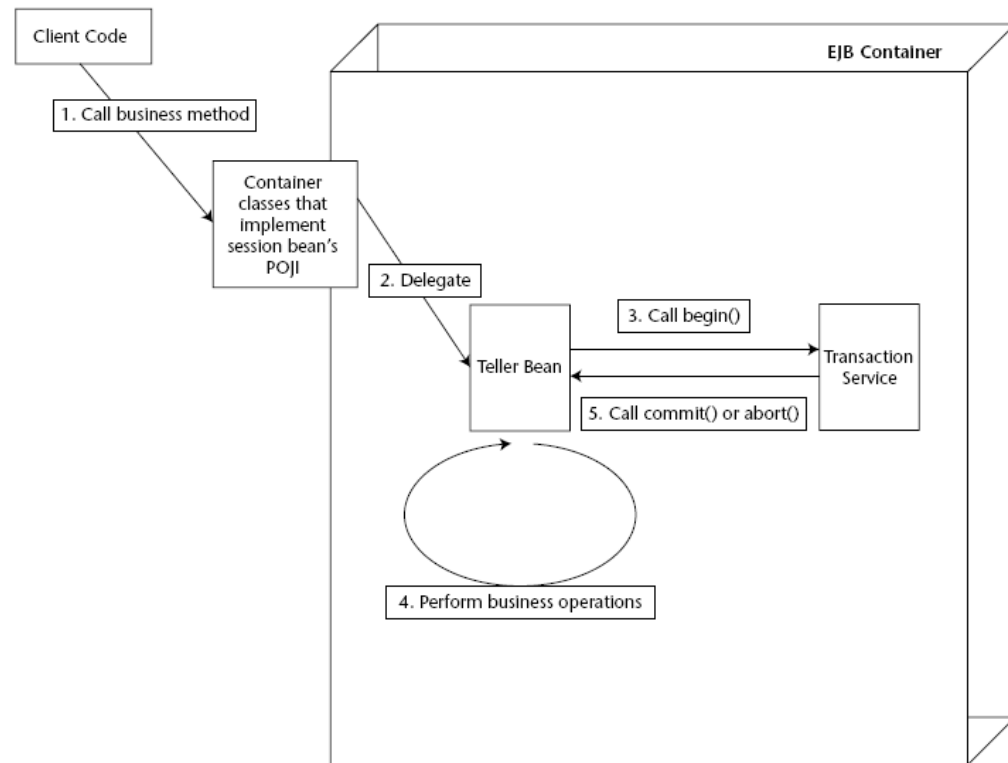


Demarcating Transaction Boundaries

- Bean Managed Transaction
- Container Managed Transaction
- Client Controlled Transactions

Bean Managed Transaction

- Teller Bean is responsible for issuing a begin statement to start the transaction, performing the transfer of funds , and issuing either a commit or abort statement

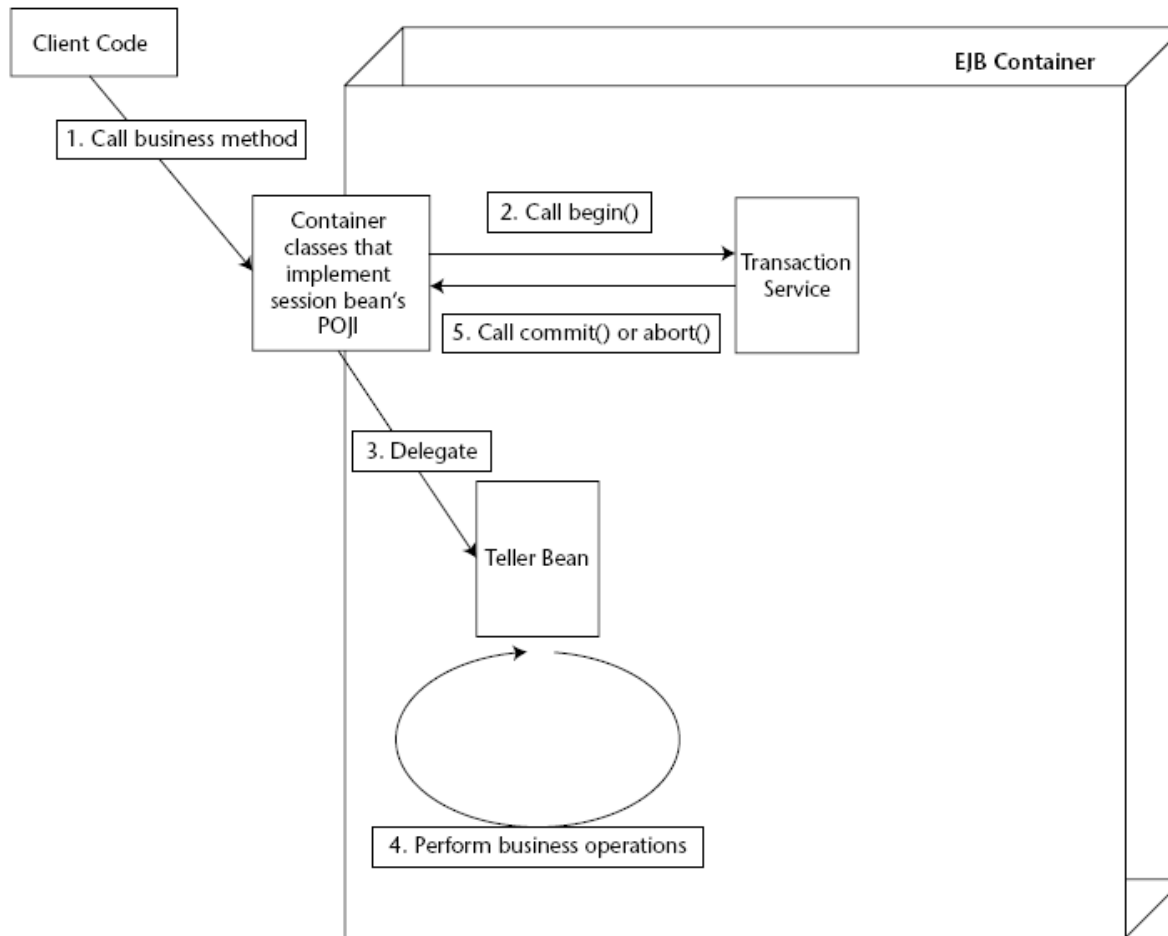


Bean-managed transactions.

Container-Managed Transaction

- Container-Managed Transactions allow for components to automatically be enlisted in transactions.
- EJB never explicitly issue a begin, commit or abort statement.
- EJB Container does it for you

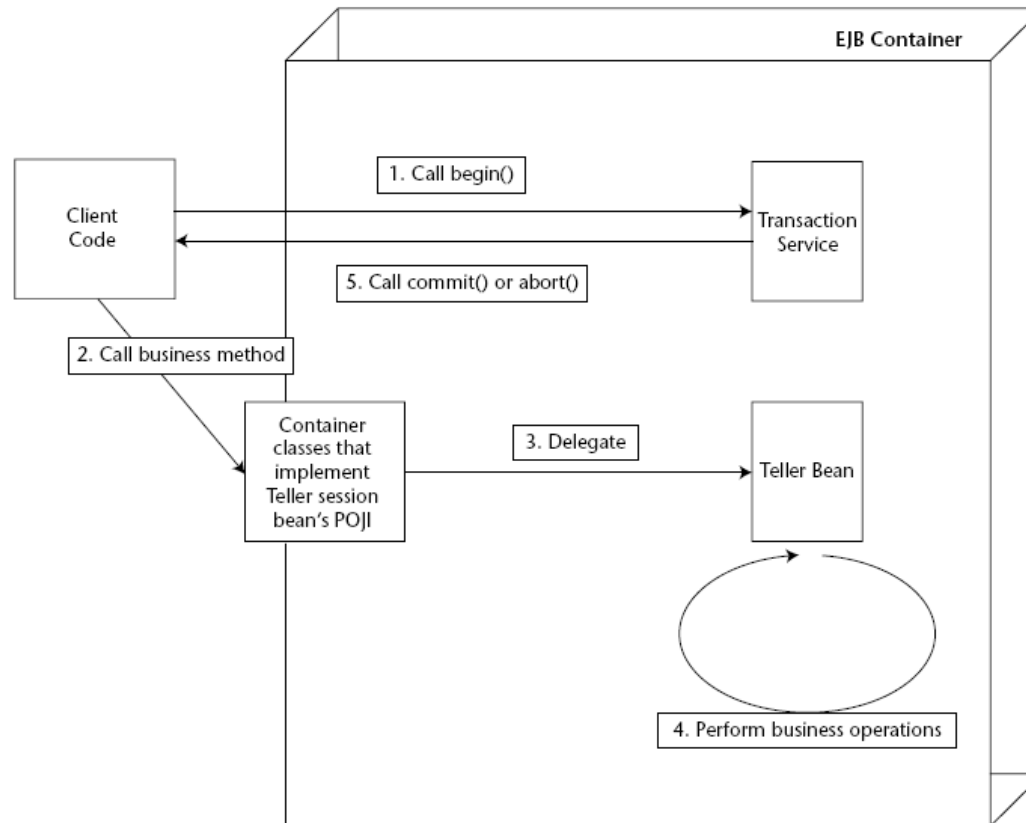
Container-Managed Transaction



Container-managed transactions.

Client-Controlled Transactions

- If you have Servlet, JSP tag library, standalone client, CORBA client or other EJB as client. you can begin and end the transaction in that caller.



Transaction Attribute Values

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Supports	None	None
	T1	T1
Mandatory	None	Error
	T1	T1
NotSupported	none	None
	T1	None
Never	none	None
	T1	Error

Container Managed Example

```
Teller.java  TellerBean.java  Client.java
package com.banu;

import javax.annotation.Resource;

@Stateless(mappedName = "TELLER")
@TransactionManagement(TransactionManagementType.CONTAINER)
public class TellerBean implements Teller {

    @Resource
    private SessionContext ctx;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void transferFunds(String fromAccount, String toAccount,
        double amount) {
        // get account info from JPA
        //get balance from fromAccount
        int balanceInFromACC = 2500;
        if (balanceInFromACC < amount) {
            ctx.setRollbackOnly();
        }
    }
}
```

↖ This will ensure that the container never commits transaction under such a condition.

Specifying Tx attributes in Deployment Descriptor

```
<enterprise-beans>
  <session>
    <display-name>TellerBean</display-name>
    <ejb-name>TellerBean</ejb-name>
    <business-remote>Teller</business-remote>
    <ejb-class>TellerBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>

<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>TellerBean</ejb-name>
      <method-name>transferFunds</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```


Bean Managed Transaction Example

- Programmatically demarcating transactions allow for more transaction control than container managed transactions do.
- To control transaction boundaries , you must use JTA interface `javax.transaction.UserTransaction`.

UserTransaction Methods

METHOD	DESCRIPTION
<code>begin()</code>	Begins a new transaction. This transaction becomes associated with the current thread.
<code>commit()</code>	Runs the two-phase commit protocol on an existing transaction associated with the current thread. Each resource manager will make its updates durable.
<code>getStatus()</code>	Retrieves the status of the transaction associated with this thread.
<code>Rollback()</code>	Forces a rollback of the transaction associated with the current thread.
<code>setRollbackOnly()</code>	Calls this to force the current transaction to roll back. This will eventually force the transaction to abort.
<code>setTransactionTimeout(int)</code>	The transaction timeout is the maximum amount of time that a transaction can run before it's aborted. This is useful for avoiding deadlock situations, when precious resources are being held by a transaction that is currently running.

Bean managed transaction example

```
@Stateless
@Remote
@TransactionManagement(TransactionManagementType.BEAN) // specify bean managed tx
public class TellerBean implements Teller {

    @Resource
    private UserTransaction tx;

    public void transferFunds(String fromAcc, String toAcc, double amt) {
        try {
            tx.begin();
            // actual code for fund transfer
            tx.commit();
        } catch (Exception ex) {
            try {
                tx.rollback();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Transaction Isolation Levels

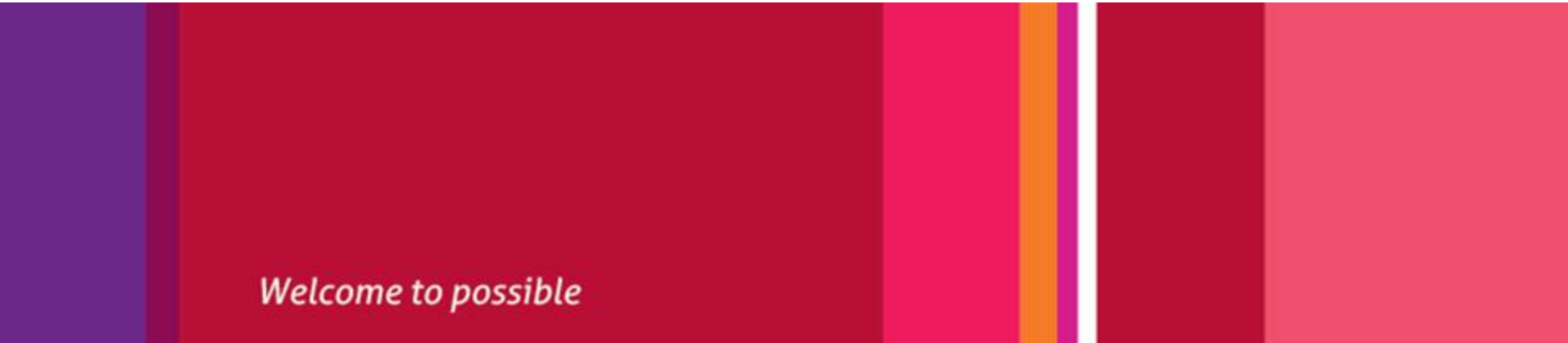
The Isolation Levels

ISOLATION LEVEL	DIRTY READS?	UNREPEATABLE READS?	PHANTOM READS?
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

EJB does not provide a way to specify isolation levels directly

We specify Isolation Levels on the resource API itself.

Example: `Connection.setIsolationLevel()` method



Welcome to possible

Name

Email

www.mindtree.com/social