**Quick Reference to**

**Enterprise JavaBeans (EJB)**

Index

[1. ORM & JPA & Hibernate/EclipseLink 2](#_Toc469072313)

[2. What is the difference between EJB, hibernate, spring and JSF? 3](#_Toc469072314)

[3. Difference between EJB 3.0 and EJB 3.1 3](#_Toc469072315)

[4. New Features of EJB 3.1 4](#_Toc469072316)

[5. persistence.xml (Sample) 6](#_Toc469072317)

[6. Difference between JTA, JPA and Plain JDBC in hibernate 6](#_Toc469072318)

[7. Java Persistence API 7](#_Toc469072319)

[JPA History 8](#_Toc469072320)

[Where to use JPA? 8](#_Toc469072321)

[JPA 2.0 8](#_Toc469072322)

[JPA 2.1 9](#_Toc469072323)

[8. EclipseLink/Examples/JPA/OutsideContainer 10](#_Toc469072324)

[JavaSE Configuration using persistence.xml 10](#_Toc469072325)

[9. JPA Transaction Types and JTA 10](#_Toc469072326)

[10. What are the different kinds of enterprise beans? 12](#_Toc469072327)

[11. What is Session Bean and when to use Session Bean? 12](#_Toc469072328)

[Types of Session Beans 12](#_Toc469072329)

[When to Use Session Beans 15](#_Toc469072330)

[12. What is Entity Bean? 15](#_Toc469072331)

# ORM & JPA & Hibernate/EclipseLink

* [JPA](http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html) is the interface while Hibernate is the implementation.

Traditionally there have been multiple Java ORM solutions:

* [Hibernate](http://hibernate.org/orm/documentation/)
* [TopLink](http://en.wikipedia.org/wiki/TopLink) (now it is EclipseLink)
* [JDO](http://en.wikipedia.org/wiki/Java_Data_Objects)

each implementation defining its own mapping definition or client API. The JPA expert group gathered the best of all these tools and so they created the Java Persistence API standard.

A standard persistence API is very convenient from a client point of view, making it relatively easy to switch one implementation with the other (although in practice it's not that simple, because on large projects you'll have to use specific non-standard features anyway).

The standard JPA has pushed Java ORM competition to a new level and this can only lead to better implementations.

Thus, many of the features originally presented in third-party persistence frameworks were incorporated into the Java Persistence API, and, as of 2006, projects like [Hibernate](https://en.wikipedia.org/wiki/Hibernate_(Java)) (version 3.2) and [TopLink Essentials](https://en.wikipedia.org/wiki/TopLink) have become themselves implementations of the Java Persistence API specification.

# What is the difference between EJB, hibernate, spring and JSF?

* (<http://stackoverflow.com/questions/4281304/what-is-the-difference-between-ejb-hibernate-spring-and-jsf>)

These are frameworks for different layers.

* **JSF** is for the view (web) layer, it's a component oriented framework (every part of a page is a component, it has state) like Wicket or Tapestry, and unlike Action frameworks like Spring MVC, Struts or Stripes
* **EJB 3.x** is a container that's part of the [JavaEE](http://download.oracle.com/javaee) stack. It does things like dependency injection and bean lifecycle management. You usually need a full JavaEE application server for EJB3
* **Spring** is also a container, but Spring can run in any java code (a simple main class, an applet, a web app or a JavaEE enterprise app). Spring can do almost everything EJB can do and a lot more, but I'd say it's most famous for dependency injection and non-intrusive transaction management
* **Hibernate** was the first big [ORM](http://en.wikipedia.org/wiki/Object-relational_mapping) (Object relational mapper) on the Java Platform, and as such has greatly inspired **JPA** (which is part of the EJB3 standard but can be used without an EJB container). I would suggest coding against JPA and only using hibernate as a provider, that way you can easily switch to EclipseLink etc.

# Difference between EJB 3.0 and EJB 3.1

* EJB 3.0 and 3.1 are two versions of the Enterprise JavaBeans specification, which is a part of the overall Java Enterprise Edition (JEE).

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Feature** | **EJB 3.1** | **EJB 3.0** |
| 1. | Bean Types | Stateless, Stateful, Singleton | Stateless and Stateful |
| 2. | Asynchronous Services | Available | Not Available |
| 3. | Global JNDI names | Part of specification | Server dependent feature |
| 4. | Timer-Service | Available | Not Available |
| 5. | Packaging Methodology | Allows EJBs to be packaged inside WARs | EJBs need to be packaged into a JAR which when packaged with application WAR gives an EAR deployable package. |
| 6. | Embeddable EJB Containers | Part of specification | Not Available |
| 7. | No-Interface View | Part of specification | Not Available |

# New Features of EJB 3.1

* EJB, the Enterprise Java Beans is a server-side technology from Sun for creating modular J2EE applications. EJB has been widely criticized for its complexity. To ease this complexity was the main focus of EJB 3.0 release in 2006. It was successful in taking away a lot of complexity and repetitiveness of EJB 2.1 and earlier releases. The major enhancement in this release was the use of annotations instead of xml descriptors for configuration of enterprise beans.

EJB 3.1, released in 2009, further attempts to ease the task of the EJB developer. Following are some of the key features of EJB 3.1,

**1. Session Bean Business Interfaces made Optional:** In earlier version of EJB, creating interfaces for enterprise beans was necessary. It was the methods described in the interface (local or remote interface) that could be accessed by the clients of the bean. Though the idea of creating interfaces brings in the benefits of loosely coupled and unit-testable code, at times it is overkill when none of these features are desired.

EJB 3.1 provides “No Interface View” feature that makes it possible to create an enterprise bean without creating any interfaces. All the public methods of the bean will be available to its clients.

**2. Singleton EJBs:** Before singleton EJB,s if we ever wanted to create an application wide cache we had either to resort to Server specific solutions or use static fields. Using static fields were not thread safe out of the box and lacked features like dependency injection, declarative transaction management, security, remoting, component life-cycle callbacks, interceptors etc. All these features are now available with Singleton Beans.

**3. Enhanced EJB Timer Service:** EJB Timer service has been enhanced to provide cron-like scheduling capabilities. Other enhancements in the Timer Service are deployment-time timer creation and Stateful Session Bean timed objects. Now it is possible to declaratively create schedules to trigger EJB methods. @Schedule annotation is used to implement such a timer.

**4. Asynchronous Services:** EJBs now have asynchronous capabilities. Now the client does not have to wait on an EJB to process a method invocation. The method call returns instantly either returning void or an instance of java.util.concurrent.Future. In case of the latter the client may at any time in future check on the status of processing by accessing returned object. This allows the EJB container to buffer method calls so as to prevent bottlenecks. The only catch here is that this capability can only be used where there is no need of an immediate result for proceeding further on the client end.

In order to create an EJB as Asynchronous, it needs to be annotated with @Asynchronous keyword. If an EJB is annotated in this way, then it makes all its methods asynchronous. If we wish to make only a few methods asynchronous, we can do that by annotating only those methods with @Asynchronous keyword.

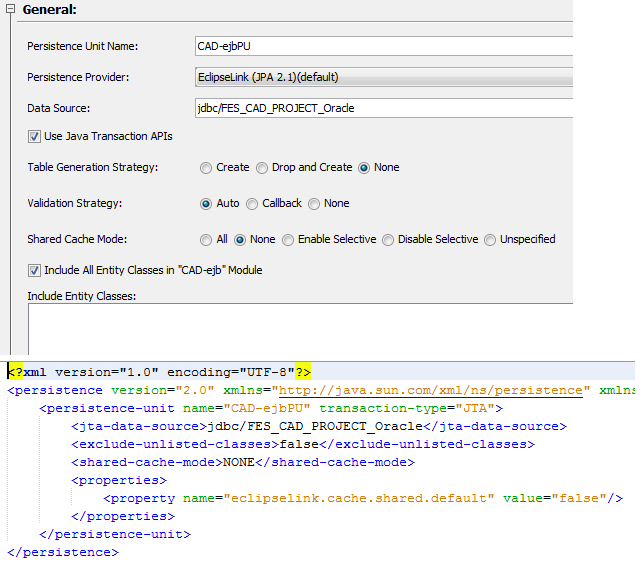
**5. Simplified Deployment:** EJB 3.1 makes it easier to package EJB classes in .WAR files. Before EJB 3.1, the process of deploying an EJB application involved packaging the web components into a .war file, the EJBs into a .jar file and then to combine them into an .ear package. It is this .ear package that was to be deployed onto the EJB container.

This process has been simplified greatly. Now from EJB3.1 EJBs can directly be placed in the WEB-INF/classes folder of the web application. Also if your application contains ejb-jar.xml configuration file, even that can go into the WEB-INF directory.

**6. Easier to unit test EJBs:** Another new feature in EJB 3.1 specification is the facility to use EJB outside an EJB container. Open source implementations of embeddable EJB containers like Open EJB, GlassFish & JBoss are available and they can directly run in Java SE environment.

# persistence.xml (Sample)

* Following is the sample persistence.xml file which is using EclipseLinks as Persistence Provider.



# Difference between JTA, JPA and Plain JDBC in hibernate

* (<http://stackoverflow.com/questions/3903477/difference-between-jta-jpa-and-plain-jdbc-in-hibernate>)
* **JDBC** is a Java standard for database connection.

JDBC is a technology for accessing databases. It is what Hibernate actually uses to perform the database operations, "under the hood". It uses JDBC to send queries to the database.

* **JPA** is a standard for Java object-relational mapping - it specifies a set of annotations and an interface -EntityManager to perform persistence operations with the mapped objects. Hibernate implements the JPA standard

JPA isolates the Java developer from the inner workings of JDBC and database operations.

Hibernate, EclipseLink, OpenJPA and Data Nucleus are famous JPA implementations.

* **JTA** is a standard for transactions, allowing for management of multiple transactions among multiple databases.

[JTA](http://en.wikipedia.org/wiki/Java_Transaction_API) is a transaction API, and it is optional in Hibernate. It handles (logically) the transaction behaviour.

**JPA utilizes JDBC** for database connections and SQL-related operations, and -optionally- utilizes JTA for delegating distributed transaction management details to it.

# Java Persistence API

* **(**[**http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html**](http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html)**)**

The Java Persistence API provides Java developers with **an object/relational mapping** facility for managing **relational data** in Java applications. Java Persistence consists of four areas:

* The Java Persistence API
* The query language
* The Java Persistence Criteria API
* Object/relational mapping metadata

(<https://www.jcp.org/en/jsr/detail?id=220>)

(<http://www.tutorialspoint.com/jpa/jpa_entity_managers.htm>)

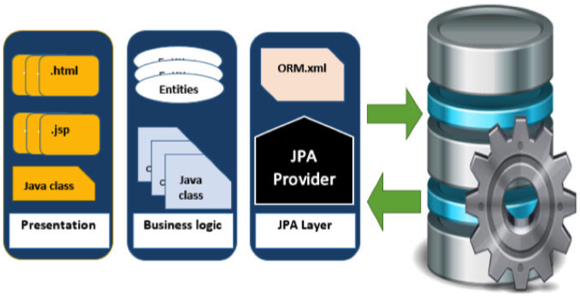
## JPA History

Earlier versions of EJB, defined persistence layer combined with business logic layer using javax.ejb.EntityBean Interface.

* While introducing EJB 3.0, the persistence layer was separated and specified as JPA 1.0 (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006 using JSR 220.
* JPA 2.0 was released with the specifications of JAVA EE6 on December 10, 2009 as a part of Java Community Process JSR 317.
* JPA 2.1 was released with the specification of JAVA EE7 on April 22, 2013 using JSR 338.

## Where to use JPA?

To reduce the burden of writing codes for relational object management, a programmer follows the ‘JPA Provider’ framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.



## JPA 2.0

Development of a new version of JPA 2.0 was started in July 2007 in the Java Community Process as JSR 317. JPA 2.0 was approved as final on 10 December 2009. The focus of JPA 2.0 was to address features that were present in some of the popular ORM vendors but could not gain consensus approval for JPA 1.0.

Main features included were:

* Expanded object-relational mapping functionality
  + support for collections of embedded objects, linked in the ORM with a [many-to-one relationship](https://en.wikipedia.org/wiki/Cardinality_(data_modeling))
  + ordered lists
  + combinations of access types
* A criteria query API
* standardization of query 'hints'[[*clarify*](https://en.wikipedia.org/wiki/Wikipedia:Please_clarify)]
* standardization of additional metadata to support DDL generation
* support for validation
* Shared object cache support.

Vendors supporting JPA 2.0:

* [Batoo JPA](https://en.wikipedia.org/wiki/Batoo_JPA)
* [DataNucleus](https://en.wikipedia.org/wiki/DataNucleus) (formerly JPOX)
* [EclipseLink](https://en.wikipedia.org/wiki/EclipseLink) (formerly [Oracle TopLink](https://en.wikipedia.org/wiki/TopLink))
* [JBoss](https://en.wikipedia.org/wiki/JBoss) with [Hibernate](https://en.wikipedia.org/wiki/Hibernate_(Java))
* [Kundera](https://github.com/impetus-opensource/Kundera)
* [ObjectDB](https://en.wikipedia.org/wiki/ObjectDB)
* [OpenJPA](https://en.wikipedia.org/wiki/OpenJPA)
* [OrientDB](https://en.wikipedia.org/wiki/OrientDB) from [Orient Technologies](https://en.wikipedia.org/w/index.php?title=Orient_Technologies&action=edit&redlink=1)
* [IBM](https://en.wikipedia.org/wiki/IBM), for [WebSphere Application Server](https://en.wikipedia.org/wiki/WebSphere_Application_Server)[[9]](https://en.wikipedia.org/wiki/Java_Persistence_API#cite_note-9)
* [Versant Corporation](https://en.wikipedia.org/wiki/Versant_Corporation) JPA (not relational, object database)[[10]](https://en.wikipedia.org/wiki/Java_Persistence_API#cite_note-10)

## JPA 2.1

Development of a new version of JPA 2.1 was started in July 2011 as JSR 338. JPA 2.1 was approved as final on 22 May 2013.

Main features included were:

* Converters - allowing custom code conversions between database and object types.
* Criteria Update/Delete - allows bulk updates and deletes through the Criteria API.
* Stored Procedures - allows queries to be defined for database stored procedures.
* Schema Generation
* Entity Graphs - allow partial or specified fetching or merging of objects.
* JPQL/Criteria enhancements - arithmetic sub-queries, generic database functions, join ON clause, TREAT option.

Vendors supporting JPA 2.1

* [DataNucleus](https://en.wikipedia.org/wiki/DataNucleus)
* [EclipseLink](https://en.wikipedia.org/wiki/EclipseLink)
* [Hibernate](https://en.wikipedia.org/wiki/Hibernate_(Java))

# EclipseLink/Examples/JPA/OutsideContainer

* (<https://wiki.eclipse.org/EclipseLink/Examples/JPA/OutsideContainer>)

**Users may also use the EntityManager API outside the container**, either through configuring a persistence.xml, or by creating a EntityManagerFactory for a given persistence unit name and properties map (includes database information etc.)

EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnitName, propertiesMap);

EntityManager em = emf.createEntityManager();

## JavaSE Configuration using persistence.xml

<persistence …>

<persistence-unit name="my-app" **transaction-type="RESOURCE\_LOCAL"**>

<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

<exclude-unlisted-classes>false</exclude-unlisted-classes>

<properties>

<property name="javax.persistence.jdbc.driver" value="oracle.jdbc.OracleDriver"/>

<property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:@localhost:1521:orcl"/>

<property name="javax.persistence.jdbc.user" value="scott"/>

<property name="javax.persistence.jdbc.password" value="tiger"/>

</properties>

</persistence-unit>

</persistence>

Note, "javax.persistence.jdbc" is used in JPA 2.0, as of EclipseLink 1.2, previously "eclipselink.jdbc" must be used.

Now the EntityManagerFactory can be instantiated using:

Persistence.createEntityManagerFactory("my-app");

# JPA Transaction Types and JTA

* (http://www.mytechnotes.biz/2012/08/jpa-transaction-types-and-jta.html)

In the persistence.xml JPA configuration file, one has to specify the *transaction type* for each persistence unit:

<persistence ... >

<persistence-unit name="..." transaction-type="RESOURCE\_LOCAL">

...

</persistence-unit>

...

</persistence>

There are two possible values: RESOURCE\_LOCAL and JTA.

#### Resource Local

* This is the typical value for a standalone application
* One is responsible for the creation of the EntityManager via the EntityManagerFactory
* Transactions are of type EntityTransaction
* One must use begin() and commit() around every transaction
* If multiple instances of EntityManager are created for the same persistence unit, multiple persistence contexts are created too. Meaning, transactions are NOT synchronized across these in the application. A solution is to use [UserTransaction](http://docs.oracle.com/javaee/6/api/javax/transaction/UserTransaction.html). Another is to avoid using multiple EntityManager for the same persistence unit.

Usage:

// Creating resources

EntityManagerFactory EMF = Persistence.createEntityManagerFactory("Standalone");

EntityManager EM = EMF.createEntityManager();

// Transaction

EntityTransaction et = EM.getTransaction();

try {

et.begin();

// Operations...

et.commit();

} catch(Exception ex) {

et.rollback();

}

// Closing resources

EM.close();

EMF.close();

#### JTA

JTA stands for Java Transaction API. It is part of the Java EE specification.

* This is the typical value for a container application
* The EntityManager is provided by the container (unique instance across the application)
* Transactions can be declared with @TransactionAttribute and will be managed automatically
* All transactions are synchronized over the application with a unique JTA transaction

Usage:

@PersistenceContext(unitName="Container")

private EntityManager EM;

@TransactionAttribute(TransactionAttributeType.REQUIRES\_NEW)

public void myMethod() throws Exception {

Item i = EM.find(Item.class, 997);

// ...

}

More about JPA/JTA transactions [here](http://en.wikibooks.org/wiki/Java_Persistence/Transactions), more about transaction subtleties [here](http://www.ibm.com/developerworks/java/library/j-ts1/index.html).

# What are the different kinds of enterprise beans?

* **Stateless session bean**- An instance of these non-persistent EJBs provides a service without storing an interaction or conversation state between methods. Any instance can be used for any client.

**Stateful session bean**- An instance of these non-persistent EJBs maintains state across methods and transactions. Each instance is associated with a particular client.

**Entity bean**- An instance of these persistent EJBs represents an object view of the data, usually rows in a database. They have a primary key as a unique identifier. Entity bean persistence can be either container-managed or bean-managed.

**Message-driven bean**- An instance of these EJBs is integrated with the Java Message Service (JMS) to provide the ability for message-driven beans to act as a standard JMS message consumer and perform asynchronous processing between the server and the JMS message producer.

# What is Session Bean and when to use Session Bean?

* (<http://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html>)

A **session bean** encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean’s methods. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

A session bean is not persistent. (That is, its data is not saved to a database.)

For code samples, see [Chapter 24, Running the Enterprise Bean Examples](http://docs.oracle.com/javaee/6/tutorial/doc/gijrb.html).

### Types of Session Beans

Session beans are of three types: stateful, stateless, and singleton.

#### Stateful Session Beans

The state of an object consists of the values of its instance variables. In a **stateful session bean**, the instance variables represent the state of a unique client/bean session. Because the client interacts (“talks”) with its bean, this state is often called the **conversational state**.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. When the client terminates, its session bean appears to terminate and is no longer associated with the client.

The state is retained for the duration of the client/bean session. If the client removes the bean, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends, there is no need to retain the state.

#### stateful-session-bean-state-management.jpg

#### Stateless Session Beans

A **stateless session bean** does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean’s instance variables may contain a state specific to that client but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained. Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. That is, the state of a stateless session bean should apply across all clients.

Because they can support multiple clients, stateless session beans can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

A stateless session bean can implement a web service, but a stateful session bean cannot.

#### Singleton Session Beans

A **singleton session bean** is instantiated once per application and exists for the lifecycle of the application. Singleton session beans are designed for circumstances in which a single enterprise bean instance is shared across and concurrently accessed by clients.

Singleton session beans offer similar functionality to stateless session beans but differ from them in that there is only one singleton session bean per application, as opposed to a pool of stateless session beans, any of which may respond to a client request. Like stateless session beans, singleton session beans can implement web service endpoints.

Singleton session beans maintain their state between client invocations but are not required to maintain their state across server crashes or shutdowns.

Applications that use a singleton session bean may specify that the singleton should be instantiated upon application startup, which allows the singleton to perform initialization tasks for the application. The singleton may perform cleanup tasks on application shutdown as well, because the singleton will operate throughout the lifecycle of the application.

### When to Use Session Beans

Stateful session beans are appropriate if any of the following conditions are true.

* The bean’s state represents the interaction between the bean and a specific client.
* The bean needs to hold information about the client across method invocations.
* The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
* Behind the scenes, the bean manages the work flow of several enterprise beans.

To improve performance, you might choose a stateless session bean if it has any of these traits.

* The bean’s state has no data for a specific client.
* In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an email that confirms an online order.
* The bean implements a web service.

Singleton session beans are appropriate in the following circumstances.

* State needs to be shared across the application.
* A single enterprise bean needs to be accessed by multiple threads concurrently.
* The application needs an enterprise bean to perform tasks upon application startup and shutdown.

The bean implements a web service.

# What is Entity Bean?

* The entity bean is used to represent data in the database. It provides an object-oriented interface to data that would normally be accessed by the JDBC or some other back-end API. More than that, entity beans provide a component model that allows bean developers to focus their attention on the business logic of the bean, while the container takes care of managing persistence, transactions, and access control.

(Entity Beans (both BMP and CMP) no longer exist in [EJB](http://www.coderanch.com/forums/f-11/ejb-jee) 3 - they have been replaced by the Persistence API.)

# Transaction Management Type in EJB

* For Detail Read,

(<http://docs.oracle.com/javaee/6/tutorial/doc/bncij.html>) and

(<https://examples.javacodegeeks.com/enterprise-java/ejb3/transactions/ejb-transaction-management-example/>) and

(<http://docs.oracle.com/javaee/6/tutorial/doc/bnciy.html>) and

(<http://docs.oracle.com/javaee/6/tutorial/doc/bnbqw.html>)

## Container-Managed Transactions

In an enterprise bean with **container-managed transaction demarcation**, the EJB container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean: session or message-driven. Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction’s boundaries. The code does not include statements that begin and end the transaction. By default, if no transaction demarcation is specified, enterprise beans use container-managed transaction demarcation.

Typically, the container begins a transaction immediately before an enterprise bean method starts and commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method.

## Bean-Managed Transactions

In **bean-managed transaction demarcation**, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your bean difficult, you should consider using bean-managed transactions.

The following pseudocode illustrates the kind of fine-grained control you can obtain with application-managed transactions. By checking various conditions, the pseudocode decides whether to start or stop certain transactions within the business method:

begin transaction

...

update table-a

...

if (condition-x)

commit transaction

else if (condition-y)

update table-b

commit transaction

else

rollback transaction

begin transaction

update table-c

commit transaction

When coding an application-managed transaction for session or message-driven beans, you must decide whether to use Java Database Connectivity or JTA transactions. The sections that follow discuss both types of transactions.

To demarcate (set a boundry) a JTA transaction, you invoke the begin, commit, and rollback methods of the javax.transaction.UserTransaction interface.

### Returning without Committing

In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction.

In a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each business method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction.

In a stateful session bean with a JDBC transaction, the JDBC connection retains the association between the bean instance and the transaction across multiple calls. If the connection is closed, the association is not retained.

### Methods Not Allowed in Bean-Managed Transactions

Do not invoke the getRollbackOnly and setRollbackOnly methods of the EJBContext interface in bean-managed transactions. These methods should be used only in container-managed transactions. For bean-managed transactions, invoke the getStatus and rollback methods of the UserTransaction interface.

# Container Managed Session Bean Example

package org.fes.pis.sessions;

import java.util.List;

import javax.ejb.Stateless;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import org.fes.pis.entities.BmwpApBudgetMst;

@Stateless

public class BmwpApBudgetMstFacade extends AbstractFacade<BmwpApBudgetMst> implements BmwpApBudgetMstFacadeLocal {

@PersistenceContext(unitName = "PIS-app-ejbPU")

private EntityManager em;

@Override

protected EntityManager getEntityManager() {

return em;

}

public BmwpApBudgetMstFacade() {

super(BmwpApBudgetMst.class);

}

@Override

public List<BmwpApBudgetMst> findAllAPwise(Integer p\_apl\_unique\_srno) {

String v\_query = " SELECT \* FROM BMWP\_AP\_BUDGET\_MST"

+ " WHERE APBM\_APL\_UNIQUE\_SRNO = " + p\_apl\_unique\_srno

+ " ORDER BY APBM\_UNIQUE\_SRNO";

return em.createNativeQuery(v\_query, BmwpApBudgetMst.class).getResultList();

}

}

# Bean (Application) Managed Session Bean Example

import java.io.Serializable;

import java.util.logging.Level;

import javax.annotation.Resource;

import javax.ejb.EJBContext;

import javax.ejb.Stateless;

import javax.ejb.TransactionManagement;

import javax.ejb.TransactionManagementType;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import javax.transaction.UserTransaction;

import org.fes.lib.utilities.LogGenerator;

import org.fes.survey.custom\_entities.System\_Properties;

import org.fes.survey.entities.\*;

@TransactionManagement(TransactionManagementType.BEAN)

@Stateless

public class TransactionJPA implements Serializable {

@PersistenceContext(unitName = "surveyPU")

private EntityManager em;

@Resource

private EJBContext context;

public EntityManager getEntityManager() {

return em;

}

System\_Properties system\_Properties = new System\_Properties();

String systemName = system\_Properties.getSystemName();

public TransactionJPA() {

}

public boolean VerifySurveyMsts(FmisHabitSurveyMst[] selectedSurveyMsts) {

boolean isTransactionComplete = false;

UserTransaction utx = context.getUserTransaction();

try {

utx.begin();

int n = selectedSurveyMsts.length;

for (int i = 0; i < n; i++) {

FmisHabitSurveyMst ent\_FmisHabitSurveyMst = em.find(FmisHabitSurveyMst.class, selectedSurveyMsts[i].getHabUniqueSrno());

ent\_FmisHabitSurveyMst.setAprvcomm(“test”);

em.merge(ent\_FmisHabitSurveyMst);

}

utx.commit();

isTransactionComplete = true;

} catch (Exception ex) {

try {

LogGenerator.generateLog(systemName, Level.SEVERE, this.getClass().getName(), "VerifySurveyMsts", null, ex);

utx.setRollbackOnly();

} catch (Exception re) {

LogGenerator.generateLog(systemName, Level.SEVERE, this.getClass().getName(), "VerifySurveyMsts", null, re);

}

} finally {

//entity manager should be closed

}

return isTransactionComplete;

}

}

# AbstractFacade Example

package org.fes.pis.sessions;

import java.util.List;

import java.util.logging.Level;

import java.util.logging.Logger;

import javax.persistence.EntityManager;

import javax.validation.ConstraintViolationException;

public abstract class AbstractFacade<T> {

private Class<T> entityClass;

public AbstractFacade(Class<T> entityClass) {

this.entityClass = entityClass;

}

protected abstract EntityManager getEntityManager();

public void create(T entity) {

try {

getEntityManager().persist(entity);

} catch (ConstraintViolationException e) {

Logger.getLogger(this.getClass().getName()).log(Level.SEVERE, e.getConstraintViolations().toString(), e);

throw e;

}

}

public void edit(T entity) {

try {

getEntityManager().merge(entity);

} catch (ConstraintViolationException e) {

Logger.getLogger(this.getClass().getName()).log(Level.SEVERE, e.getConstraintViolations().toString(), e);

throw e;

}

}

public void remove(T entity) {

getEntityManager().remove(getEntityManager().merge(entity));

}

public T find(Object id) {

return getEntityManager().find(entityClass, id);

}

public List<T> findAll() {

javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();

cq.select(cq.from(entityClass));

return getEntityManager().createQuery(cq).getResultList();

}

public List<T> findRange(int[] range) {

javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();

cq.select(cq.from(entityClass));

javax.persistence.Query q = getEntityManager().createQuery(cq);

q.setMaxResults(range[1] - range[0]);

q.setFirstResult(range[0]);

return q.getResultList();

}

public int count() {

javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();

javax.persistence.criteria.Root<T> rt = cq.from(entityClass);

cq.select(getEntityManager().getCriteriaBuilder().count(rt));

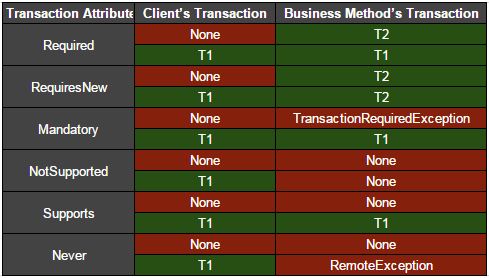
javax.persistence.Query q = getEntityManager().createQuery(cq);

return ((Long) q.getSingleResult()).intValue();

}

}

# Container Managed Transaction (CMT) TransactionAttributes

[](http://examples.javacodegeeks.com/wp-content/uploads/2015/04/CMT1.jpg)

# Setting Transaction Attributes

* Transaction attributes are specified by decorating the enterprise bean class or method with a [javax.ejb.TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html) annotation and setting it to one of the [javax.ejb.TransactionAttributeType](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html) constants.  
  By default, if a [TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html) annotation is not specified for a method of an enterprise bean with container-managed transaction demarcation, the value of the transaction attribute for the method is defined to be [REQUIRED](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#REQUIRED).

If you decorate the enterprise bean class with [@TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html), the specified [TransactionAttributeType](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html) is applied to all the business methods in the class. Decorating a business method with [@TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html) applies the [TransactionAttributeType](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html) only to that method. If a [@TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html) annotation decorates both the class and the method, the method [TransactionAttributeType](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html) overrides the class [TransactionAttributeType](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html).

The following code snippet demonstrates how to use the [@TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html) annotation:

package com.javacodegeeks.example.beans;

import javax.ejb.Stateless;

import javax.ejb.TransactionAttribute;

import javax.ejb.TransactionAttributeType;

@Stateless

@TransactionAttribute(TransactionAttributeType.NOT\_SUPPORTED)

public class SampleBean {

...

@TransactionAttribute(TransactionAttributeType.REQUIRES\_NEW)

public void firstMethod() {...}

@TransactionAttribute(TransactionAttributeType.MANDATORY)

public void secondMethod() {...}

public void thirdMethod() {...}

}

In this example, the SampleBean class’s transaction attribute has been set to [NotSupported](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#NOT_SUPPORTED), firstMethod has been set to [RequiresNew](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttributeType.html#REQUIRES_NEW), and secondMethod has been set to Mandatory . Because a [@TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html) set on a method overrides the class [@TransactionAttribute](https://docs.oracle.com/javaee/6/api/javax/ejb/TransactionAttribute.html), calls to firstMethod will create a new transaction, and calls to secondMethod must use the transaction of the client. Calls to thirdMethod do not take place within a transaction.

# Entity Class Example

import java.io.Serializable;

import javax.persistence.\*;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import javax.xml.bind.annotation.XmlRootElement;

import javax.xml.bind.annotation.XmlTransient;

@Entity

@Table(name = "FHRD\_DESIGMST")

@XmlRootElement

@NamedQueries({

@NamedQuery(name = "FhrdDesigmst.findAll", query = "SELECT f FROM FhrdDesigmst f"),…

@NamedQuery(name = "FhrdDesigmst.findByMaxTransDate", query = "SELECT f FROM FhrdDesigmst f WHERE f.maxTransDate = :maxTransDate")})

public class FhrdDesigmst implements Serializable {

private static final long serialVersionUID = 1L;

@Basic(optional = false)

@NotNull

@Column(name = "DESIG\_SRNO")

private int desigSrno;

@Basic(optional = false)

@NotNull

@Size(min = 1, max = 100)

@Column(name = "DESIG\_DESC")

private String desigDesc;

@ManyToOne(optional = false)

private SystOrgUnitMst oumUnitSrno;

public FhrdDesigmst() {

}

public FhrdDesigmst(BigDecimal dsgmSrgKey) {

this.dsgmSrgKey = dsgmSrgKey;

}

public FhrdDesigmst(BigDecimal dsgmSrgKey, int desigSrno, String desigDesc, int createby, Date createdt, Date startDate) {

this.dsgmSrgKey = dsgmSrgKey;

…

}

…

@XmlTransient

public Collection<FhrdEmpContractMst> getFhrdEmpContractMstCollection() {

return fhrdEmpContractMstCollection;

}

@Override

public String toString() {

return "org.fes.pis.entities.FhrdDesigmst[ dsgmSrgKey=" + dsgmSrgKey + " ]";

}

}

# References

* <http://www.eclipse.org/eclipselink/documentation/>
* <http://www.eclipse.org/eclipselink/documentation/2.6/concepts/toc.htm>