**Java Persistence API**

**EntityManagerFactory:**

An entity manager factory provides entity manager instances, all instances are configured to connect to same database. You can prepare several entity manager factories to connect several databases.

**EntityManager**

The EntityManager API is used to connect the database in a particular unit of work. It is used to create, update and remove the entity instances, to find the entities by their primary key identity, and to query over all entities.

EntityManger are two types

**Application-managed**: Entity managers are created when an application directly requests one from an entity manager factory. With application-managed entity managers, the application is responsible for opening or closing entity managers and involving the entity manager in transactions. This type of entity manager is most appropriate for use in standalone applications that don’t run within a Java EE container.

**Persistence Context**

A persistence context is set of entity instances in which for any persistence entity identity there is a unique entity instance. Within persistence context the entity instances and life cycle is managed by particular entity manager.

**Persistence Unit**

The set of entity types that can be managed by given entity manager is defined by persistence unit. A persistence unit defines the set of classes that are related or grouped by the application.

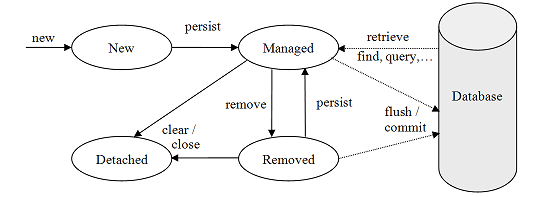
**Object states**:

**Transient objects**: Objects created through new operator are not immediately persistent. Their state is transient which means they are not associated with any database table row and so their state is lost as soon as they are no longer referenced by any other object.

**Persistent objects**: Persistent entity is entity instance with a database identity. Persistent instances may be objects instantiated by application and made persistence by calling one of the methods of Persistence manager or A persistence instance may be instance retrieved from database by execution of query.

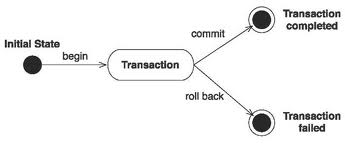
**Removed objects**: Entity instances which are having removing state. Entity instances will become removed instances by calling one of the methods of Persistence manager. Its not good to reuse removed objects.

**Detached objects**: Entity instances which are having detached instances. Persistence instances will become detached instances once persistence context closed. Detached instances will become persistent instances by calling one of the methods of persistence manager.



**Transactions**

Databases implement the notion of unit of work as database transaction. A database transaction groups the database operations-that is SQL Operations. All SQL statements execute inside transaction; there is no way to send SQL statements outside of a database transaction. A transaction is guaranteed to end in one of two ways. It is either completely commits or completely roll back. Database transactions are atomic.



**Programmatic transaction demarcation**

In this approach, you need to start and commit / rollback the transactions using following methods.

* Javax.persistence.EntityTransaction
* Javax.transaction.UserTransaction.

You need to write SQL Statements between transaction boundaries.

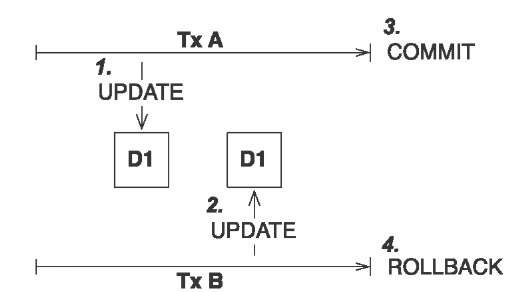
**Declarative transaction demarcation**

Declare annotations on methods. The application deployer and runtime environment can handle this type of annotations. The standard container that provides declarative transaction services in Java is an EJB container.

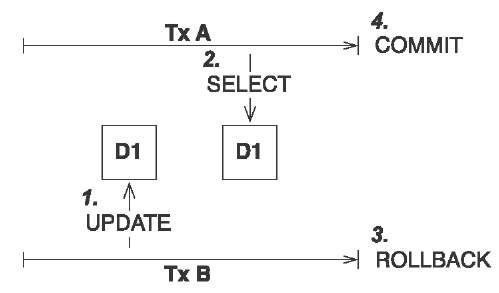
**Controlling concurrent access**

**Transaction isolation issues**

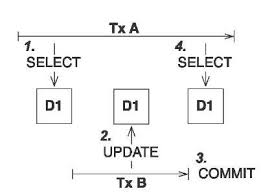
**Lost Update: Two transactions update the same data without locking.**



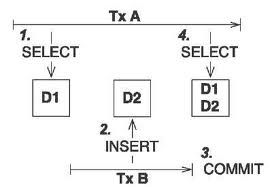
**Dirty read: Transaction reads uncommitted data.**



**Unrepeatable read: Transaction A executes two no repeatable reads**



**Phantom read: Transaction A reads new data in the second select**



Hibernate provides automatic version feature for optimistic locking.

**Enable versioning in hibernate**

Each entity has version property which can be integer or timestamp. Hibernate increments the version property for each modification, compares versions automatically and throws exceptions if a conflict is detected.

**JPA Annotations**

**Entity**: It represents persistence class. This is applied to the entity class.

* **Name**: Entity name. Default value is unqualified name of entity class. This is used to refer to the entity in queries. The name must not be reserved literal.

**Embeddable**: Defines class whose instances are stored as intrinsic part of owning entity and share the identity of the entity. Each of the properties of embedded object is mapped to database table for entity.

**Embedded**: Specifies the field of an entity whose value is an instance of embeddable class.

**AttributeOverrides**: Used to override mappings of multiple properties.

**AttributeOverride**: Used to override the mapping of property.May be applied on embedded property to override basic mapping.

**Table**: Specifies the primary table for the annotated entity. Additional tables may be specified using SecondaryTable or SecondaryTables annotation.

* **Name**: Name of the table. Default to the entity name.
* **Catalog**: The catalog of the table.

**Id**: Specifies the primary key of entity.

**GeneratedValue**: Provides for the specification of generation strategies for the values of primary keys.

* **Strategy**: the primary key generation strategy that the persistence provider must use to generate annotated entity primary key.

**Column**: It is used to specify mapped column for persistence property or field.

* **Name** : Name of column
* **Unique**: To define whether column is unique key.
* **Nullable**: Whether database column is nullable
* **Insertable**: Whether the column is included in SQL Insert statements generated by persistence provider.
* **Updatable**: Whether the column is included in SQL Update statements generated by persistence provider
* **Table**: The Name of the table that contains the column. If it is absent the column assumed to be in primary table.
* **Length**: length of the column
* **Precision**: Precision for a decimal column.
* **Scale**: The scale for decimal column.

**OneToOne**: Defines a single-valued association to another entity that has one-to-one multiplicity. It is not normally necessary to specify the associated target entity explicitly since it can be inferred from the type of object being referenced.

* **targetEntity**: entity class that is target of association.
* **Cascade**: The operations must be cascade to target of the association
* **Fetch**: whether the association should be lazily loaded or must eagerly fetched.
* **Optional**: Whether the association is optional.
* **mappedBy**: It requires in case of bidirectional association. The field that owns the relationship. This element is only specified on the inverse (non-owning) side of association.
* **orphanRemoval**: Whether to apply remove operation to entities that have been removed from the relationship and cascade remove operation to those entities.

**OneToMany**: Defines many-valued association with one-to-many multiplicity. If the collection is defined using generics to specify the element type, the associated target entity need not be specified; otherwise target entity class must be specified.

* **mappedBy**: : It requires in case of bidirectional association. The field that owns the relationship. This element is only specified on the inverse (non-owning) side of association.

**ManyToOne**: Defines single-valued association with many-to-one multiplicity. It is not normally necessary to specify the associated target entity explicitly since it can be inferred from the type of object being referenced

* **mappedBy**: It requires in case of bidirectional association. The field that owns the relationship. This element is only specified on the inverse (non-owning) side of association.

**ManyToMany**: Defines many-valued association with many-to-many multiplicity. Every many-to-many association has two sides, the owning side and non-owning or inverse side. The join table specified on the owning side

* **mappedBy**: : It requires in case of bidirectional association. The field that owns the relationship. This element is only specified on the inverse (non-owning) side of association.

**JoinColumn** : Specifies a column for joining an entity association or element collection. If the join is for OneToOne or ManyToOne mapping using foreign key strategy, the foreign key column is in the table of source entity. If the join is for OneToMany unidirectional mapping using foreign key mapping strategy, the foreign key is in the table of target entity.

* **Name**: Name of foreign key column
* **referencedColumnName**: Name of the column referenced by this foreign key column. When used with unidirectional OneToMany foreign key mapping, the reference column is in table of source entity, and in remaining associations the reference column is in table of target entity.
* **Unique**: Whether the property is unique key.
* **Nullable**: Whether foreign key column is nullable.
* **Insertable**: Whether the column is included in SQL Insert statements generated by persistence provider.
* **Updatable**: Whether the column is included in SQL Update statements generated by persistence provider