The Persistence Life Cycle, the EntityManager Interface, and Working with Detached State

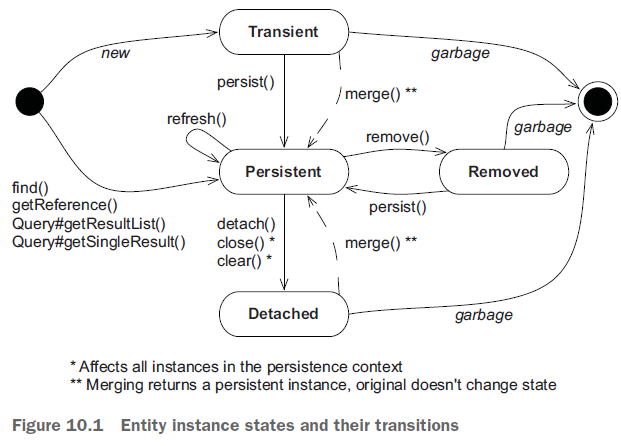
# The Persistence Life Cycle

## Basic Terminology

* **the *persistence life cycle***: the states an entity instance goes through during its life.
* ***unit of work*:** a set of (possibly) state-changing operations considered one (usually atomic) group.

## Entity Instance State

JPA defines four states, hiding the complexity of Hibernate’s internal implementation from the client code:



We discuss this chart in this chapter

### Transient State

Instances created with the new Java operator are *transient*, which means their state is lost and garbage-collected as soon as they’re no longer referenced.

Hibernate doesn’t provide any rollback functionality for transient instances; if you modify the price of a transient Item, you can’t automatically undo the change.

For an entity instance to transition from transient **to persistent state**, **to become managed,** requires either

* a call to the EntityManager#persist() method
* or the creation of a reference from an already-persistent instance and enabled cascading of state for that mapped association.

### Persistent State

**A *persistent* entity instance:**

* **has a representation in the database**.
* It’s **stored** in the database— **or** it **will be stored** when the unit of work completes.
* **It’s an instance with a database identity**, as defined in chapter 4 section2; its database identifier is set to the primary key value of the database representation.

The scenarios in which an instance could be in a persistent state:

* The application may have created instances and then made them persistent by calling EntityManager#persist().
* There may be instances that became persistent when the application created a reference to the object from another persistent instance that the JPA provider already manages.
* A persistent entity instance may be an instance retrieved from the database by execution of a query, **by an identifier lookup**, **or** by **navigating the object graph starting from another persistent instance.**

**Persistent instances are always associated with a persistence context.**

### Removed State

You can delete a persistent entity instance from the database in several ways:

* you can remove it with EntityManager#remove().
* It may also become available for deletion if you remove a reference to it from a mapped collection with *orphan* *removal* enabled.

An entity instance is then in the ***removed state*: the provider will delete it at the end of a unit of work.**

You should discard any references you may hold to it in the application after you finish working with it—for example, after you’ve rendered the removal confirmation screen your users see.

### Detached State

To understand *detached* entity instances, consider loading an instance. You call EntityManager#find() to retrieve an entity instance by its (known) identifier. **Then you end your unit of work and close the persistence context**. The application still has a *handle*—a reference to the instance you loaded. It’s now in the detached state, and the data is becoming stale. You could:

* discard the reference and let the garbage collector reclaim the memory.
* **Or, you could continue working with the data in the detached state and later call the merge()** method to save your modifications **in a new unit of work.**

# The Persistence Context

In a Java Persistence application, **an EntityManager** **has a persistence context.**

* You create a persistence context when you call EntityManagerFactory#createEntityManager().
* The context is closed when you call EntityManager#close().
* In JPA terminology, **this is an** ***application-managed* persistence context**; your application defines the scope of the persistence context, demarcating the unit of work.
* The persistence context monitors and manages all entities in persistent state.
* The persistence context **allows** the persistence engine to perform ***automatic dirty checking*,** detecting which entity instances the application modified
* The provider then synchronizes with the database the state of instances monitored by a persistence context, either automatically or on demand.
* Typically, when a unit of work completes, the provider **propagates state** held in memory **to the database** through the execution of SQL INSERT, UPDATE, and DELETE statements (all part of the Data Modification Language [DML]).
* This *flushing* procedure may also occur at other times. For example, Hibernate may synchronize with the database before execution of a query. This ensures that queries are aware of changes made earlier during the unit of work.???
* The persistence context acts as **a *first-level cache***; it remembers all entity instances you’ve handled **in a particular unit of work**. For example, if you ask Hibernate to load an entity instance using a primary key value (a lookup by identifier), Hibernate can first check the current unit of work in the persistence context. If Hibernate finds the entity instance in the persistence context, no database hit occurs—this is a repeatable read for an application. Consecutive em.find(Item.class, ITEM\_ID) calls with the same persistence context will yield the same result.
* **This cache also affects results of** arbitrary queries, executed for example with the **javax.persistence.Query API**. Hibernate reads the SQL result set of a query and transforms it into entity instances. This process first tries to resolve every entity instance in the persistence context by identifier lookup. Only if an instance with the same identifier value can’t be found in the current persistence context does Hibernate read the rest of the data from the result-set row. **Hibernate ignores any potentially newer data in the result set, due to read-committed transaction isolation at the database level,** if the entity instance is already present in the persistence context.
* **The persistence context cache is always on—it can’t be turned off.**
* The persistence context provides a *guaranteed scope of object identity*; **in the scope of a single persistence context**, **only one instance represents a particular database row.** Consider the comparison of references **entityA == entityB**. This is true only if both are references to the same Java instance on the heap. Now, consider the comparison **entityA.getId().equals(entityB.getId())**. This is true if both have the same database identifier value. Within one persistence context, Hibernate guarantees that **both comparisons will yield the same result**. This solves one of the fundamental O/R mismatch problems we introduced in section 2.3 of chapter 1.

The persistence context then ensures the following:

* **The persistence layer isn’t vulnerable to stack overflows in the case of circular references in an object graph.**
* There can never be conflicting representations of the same database row at the end of a unit of work. The provider can safely write all changes made to an entity instance to the database.
* Likewise, changes made in a particular persistence context are always immediately visible to all other code executed inside that unit of work and its persistence context. **JPA guarantees repeatable entity-instance reads.**

Would process-scoped identity be better?

For a typical web or enterprise application, persistence context-scoped identity is preferred. Process-scoped identity, where only one in-memory instance represents the row in the entire process (JVM), would offer some potential advantages in terms of cache utilization. In a pervasively multithreaded application, though, the cost of always synchronizing shared access to persistent instances in a global identity map is too high a price to pay. It’s simpler and more scalable to have each thread work with a distinct copy of the data in each persistence context.

# The EntityManager Interface

Any transparent persistence tool includes a persistence manager API. This **persistence manager** usually **provides** services for **basic CRUD** (create, read, update, delete) operations, **query execution**, and **controlling the persistence context**. **In Java Persistence** applications, **the main interface** you interact with **is the EntityManager**, to create units of work.

## The Canonical Unit of Work

you get an EntityManager by calling EntityManagerFactory#createEntityManager()

Your application code shares the **EntityManagerFactory, representing one persistence unit**, **or one logical database**. Most applications have only one shared EntityManagerFactory.

* The PersistenceUnit defines the **set of entity classes** along **with their configuration**, and it represents **a logical grouping of these entities that the entity manager manages**. We can create a persistence unit by creating a persistence.xml file or extending the PersistenceUnitInfo interface.

You use the **EntityManager** for **a single unit of work** **in a single thread**, **and it’s**

**inexpensive to create**. The following listing shows the canonical, typical form of a unit

of work. (look at it as a pseudo code)

The TM class is a convenience class bundled with the example code of this book. Here it simplifies the lookup of the standard UserTransaction API in JNDI. The JPA class provides convenient access to the shared EntityManagerFactory.)

EntityManager em = null;

UserTransaction tx = TM.getUserTransaction();

try {

tx.begin();

em = JPA.createEntityManager();

// ...

tx.commit();

} catch (Exception ex) {

// Transaction rollback, exception handling

// ...

} finally {

if (em != null && em.isOpen())

em.close();

}

!note

Everything between tx.begin() and tx.commit() occurs in one transaction. For now, keep in mind that all database operations in transaction scope, such as the SQL statements executed by Hibernate, completely either succeed or fail. Don’t worry too much about the transaction code for now; we’ll read more about concurrency control in the next chapter. We’ll look at the same example again with a focus on the transaction and exception-handling code. Don’t write empty catch clauses in your code, though—you’ll have to roll back the transaction and handle exceptions.

Creating an EntityManager starts its persistence context. Hibernate won’t access the database until necessary; the EntityManager doesn’t obtain a JDBC Connection from the pool until SQL statements have to be executed. You can create and close an EntityManager without hitting the database. Hibernate executes SQL statements when you look up or query data and when it flushes changes detected by the persistence context to the database. Hibernate joins the in-progress system transaction when an EntityManager is created and waits for the transaction to commit. When

Hibernate is notified (by the JTA engine) of the commit, it performs dirty checking of the persistence context and synchronizes with the database. You can also force dirty checking synchronization manually by calling EntityManager#flush() at any time during a transaction.

You decide the scope of the persistence context by choosing when to close() the EntityManager. You have to close the persistence context at some point, so always place the close() call in a finally block.

How long should the persistence context be open? Let’s assume for the following examples that you’re writing a server, and each client request will be processed with one persistence context and system transaction in a multithreaded environment. If you’re familiar with servlets, imagine the code in the code above embedded in a servlet’s service() method. Within this unit of work, you access the EntityManager to load and store data.

## Making data Persistent