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

**To prevent unexpected behavior when there is a difference between hibernate and JPA, you stick with either the EntityManager API or the Session API for flushing, persisting, transactions, etc.**

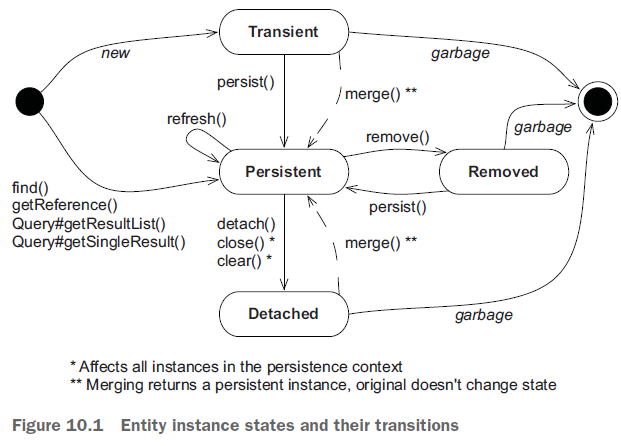
# The Persistence Life Cycle

## 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

Let’s create a new instance of an entity and bring it from transient into persistent state:

Item item = new Item();

item.setName("Some Item");

em.persist(item);

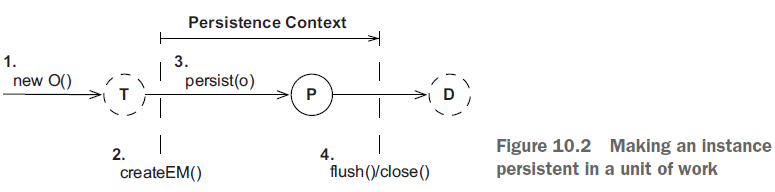
Long ITEM\_ID = item.getId();

* A new **transient Item** is instantiated as usual. Of course**, you may also instantiate it before creating the EntityManager**
* A call to persist() makes the transient instance of Item persistent. It’s now managed by and associated with the current persistence context.
* To store the Item instance in the database, Hibernate has to execute an SQL INSERT statement. When the transaction of this unit of work commits, Hibernate flushes the persistence context, and the INSERT occurs at that time. Hibernate may even batch the INSERT at the JDBC level with other statements
* **When you call persist(), only the identifier value of the Item is assigned.** **Alternatively, if your identifier generator isn’t pre-insert, the INSERT statement will be executed immediately when persist() is called.** You may want to review section 4.2.5.

Note from hands-on practice:

*Let’s say that your id is generated b a database level sequence, when calling the persist method, hibernate will only query the database to find the next value and assigns the id. On the transaction commit is when the insert statement is performed*.

You can see the same unit of work and how the Item instances changes state:



Detecting entity state using the identifier

Sometimes you need to know whether an entity instance is transient, persistent, or detached.

* **An entity instance is in persistent state if EntityManager#contains(e) returns true.**
* **It’s in transient state if PersistenceUnitUtil#getIdentifier(e) returns null:**

***I set the id on a transient entity and it returned the id. I think(through testing) that when your entity has an Id that you set yourself, the getIdentifier() method and hibernate in general will think of it as a detached entity. Because when I tried to persist this entity, it said that you can’t persist a detached entity.***

I also saw that You can call this method even before a persistence context is created.

* **It’s in detached state if it’s not persistent, and Persistence-UnitUtil#getIdentifier(e)** **returns the value of the entity’s identifier property.**

You can get to the PersistenceUnitUtil from the EntityManagerFactory. There are two issues to look out for. First, be aware that the identifier value may not be assigned and available until the persistence context is flushed. Second, Hibernate (unlike some other JPA providers) never returns null from PersistenceUnitUtil#getIdentifier() if your identifier property is a primitive (a long and not a Long).

It’s better (but not required) to fully initialize the Item instance before managing it with a persistence context. **The SQL INSERT statement contains the values that were held by the instance at the point when persist() was called.** If you don’t set the name of the Item before making it persistent, a NOT NULL constraint may be violated. **You can modify the Item after calling persist(), and your changes will be propagated to the database with** **an additional SQL UPDATE statement.**

If one of the INSERT or UPDATE statements made when flushing fails, Hibernate causes a rollback of changes made to persistent instances in this transaction **at the database level.** **But Hibernate doesn’t roll back in-memory changes to persistent instances.** If you change the Item#name after persist(), a commit failure won’t roll back to the old name. This is reasonable because **a failure of a transaction is normally non-recoverable**, and **you have to discard the failed persistence context and Entity-Manager immediately.** We’ll discuss exception handling in the next chapter.

Next, you load and modify the stored data.

## Retrieving and Modifying Persistent Data

You can retrieve persistent instances from the database with the EntityManager. For the next example, we assume you’ve kept the identifier value of the Item stored in the previous section somewhere and are now looking up the same instance **in a new unit of work by identifier**:

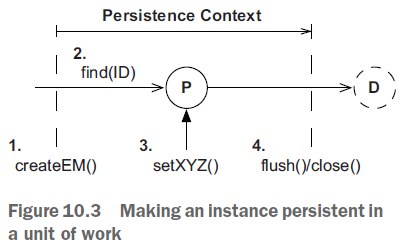
Item item = em.find(Item.class, ITEM\_ID); 🡪 **Hits database if not already in persistence context**

if (item != null)

item.setName("New Name");

* **You don’t need to cast the returned value of the find() operation**; it’s a generic method, and its return type is set as a side effect of the first parameter.
* **The retrieved entity instance is in persistent state**, and you can now modify it inside the unit of work.
* If **no persistent instance** with the given identifier value can be found, find() **returns null**
* The find() operation always hits the database if there was no hit for the given entity type and identifier **in the persistence context cache**.
* The entity instance is always initialized during loading You can expect to have all of its **values available later in detached state**: for example, when rendering a screen **after you close the persistence context**.
* (Hibernate may not hit the database if its optional second-level cache is enabled; we’ll discuss this shared cache in section 20.2.)

Figure 10.3 shows this transition graphically:



You can modify the Item instance, and the persistence context will detect these changes and record them in the database automatically. When Hibernate flushes the persistence context during commit, it executes the necessary SQL DML statements to synchronize the changes with the database.

* Hibernate propagates state changes to the database as late as possible, toward the end of the transaction. **DML statements usually create locks in the database that are held until the transaction completes, so Hibernate keeps the lock duration in the database as short as possible.**
* Hibernate writes the new Item#name to the database with an SQL UPDATE. By **default, Hibernate includes all columns of the mapped ITEM table in the SQL UPDATE statement**, updating unchanged columns to their old values. **Hence, Hibernate can generate these basic SQL statements at startup, not at runtime**.
* If you want to include **only modified** (**or non-nullable for INSERT**) columns in SQL statements, you can enable dynamic SQL generation as discussed in section 4.3.2.

Hibernate detects the changed name by **comparing the Item with a snapshot copy it**

**took before,** when the Item was loaded from the database. If your Item is different

from the snapshot, an UPDATE is necessary. **This snapshot in the persistence context**

**consumes memory**. **Dirty checking with snapshots can also be time consuming,**

**because Hibernate has to compare all instances in the persistence context with their**

**snapshot during flushing.**

* You may want to customize how Hibernate detects dirty state, using an extension

point. **Set the property hibernate.entity\_dirtiness\_strategy in your persistence.**xml configuration file to a class name that implements org.hibernate.CustomEntityDirtinessStrategy. See the Javadoc of this interface for more information.

* org.hibernate.Interceptor is another extension point used to customize dirty

checking, by implementing its findDirty() method. You can find an example interceptor

in section 13.2.2.

We mentioned earlier that the persistence context enables repeatable reads of

entity instances and provides an object-identity guarantee:

Item itemA = em.find(Item.class, ITEM\_ID);

Item itemB = em.find(Item.class, ITEM\_ID);

assertTrue(itemA == itemB);

assertTrue(itemA.equals(itemB));

assertTrue(itemA.getId().equals(itemB.getId()));

The first find() operation hits the database and retrieves the Item instance with a SELECT statement. The second find() is resolved in the persistence context, and the same cached Item instance is returned.

**Sometimes you need an entity instance but you don’t want to hit the database:**

### Getting a Reference

If you don’t want to hit the database when loading an entity instance, because you

aren’t sure you need a fully initialized instance, you can tell the EntityManager to

attempt the retrieval of a hollow placeholder—a proxy:

Item item1 = em.getReference(Item.class, ITEM\_ID);

PersistenceUnitUtil persistenceUtil = JPA.getEntityManagerFactory().getPersistenceUnitUtil();

assertFalse(persistenceUtil.isLoaded(item1));

// assertEquals(item1.getName(), "Some Item");

// Hibernate.initialize(item1);

tx.commit();

em.close();

assertEquals(item1.getName(), "Some Item");

* **If the persistence context already contains an Item with the given identifier, that Item instance is returned by getReference() without hitting the database.** Its type will be Item and no proxy will be created.
* Furthermore, if *no* persistent instance with that identifier is currently managed, **Hibernate produces a hollow placeholder: a proxy.** **This means getReference() won’t access the database, and it doesn’t return null, unlike find().** And also, the value of entityManager.contains will be true for this entity, so it’s safe to say that now the entity is in the persistent state.
* **JPA offers PersistenceUnitUtil helper methods such as isLoaded() to detect whether you’re working with an uninitialized proxy.**
* As soon as you call any method such as Item#getName() on the proxy, a SELECT is executed to fully initialize the placeholder.
* The exception to this rule **is a mapped database identifier getter method**, such as getId()even if the id does not actually correspond to a DB entity. A proxy may look like the real thing, **but it’s only a placeholder carrying the identifier value of the entity instance it represents**. If the database record no longer exists **when the proxy is initialized**, an **EntityNotFoundException** is thrown. Note that the exception can be thrown when Item#getName() is called.
* Hibernate has a convenient static initialize() method that loads the proxy’s data:

Hibernate.initialize(proxy)

* After the persistence context is closed, **item is in detached state.** If you don’t initialize the proxy while the persistence context is still open, you get a **LazyInitialization-Exception** if you access the proxy. You can’t load data on demand once the persistence context is closed. The solution is simple: load the data before you close the persistence context.

**If you want to remove the state of an entity instance from the database, you have to make it transient:**

## Making a Data Transient/removing an instance

To make an entity instance transient and delete its database representation, call the

remove() method on the EntityManager:

Item item = em.find(Item.class, ITEM\_ID);

//Item item = em.getReference(Item.class, ITEM\_ID);

em.remove(item);

assertFalse(em.contains(item));

// em.persist(item);

assertNull(item.getId());

tx.commit();

em.close();

* If you call find(), Hibernate executes a SELECT to load the Item. If you call getReference(), Hibernate attempts to avoid the SELECT and returns a proxy.
* Calling remove() **queues the entity instance for deletion** **when the unit of work completes;** it’s now in *removed* state.
* **If remove() is called on a proxy, Hibernate executes a SELECT to load the data. An entity instance must be fully initialized during life cycle transitions. You may have life cycle callback methods or an entity listener enabled (see section 13.2), and the instance must pass through these interceptors to complete its full life cycle.**

*A note from hands-on testing on this point:*

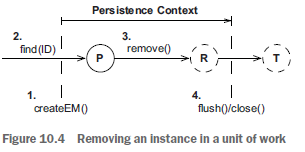
*When you call the remove method on a lazy proxy(getReference), it will still be in the persistence context even after the transaction commit. With actual persistent entities though, after the remove method is called, the entity will no longer be available in the persistence context even before the transaction commit. How?? I’ll tell you how:*

*I tested it once again, and this time I registered an interceptor for when an entity id deleted. This time upon calling the remove method, a select query was done and the instance was initialized and the interceptor was called. And after the remove method, the entity was no longer in the persistence context.*

*And when you register an interceptor, because you have all the events implemented, you’ll have the entity loaded even if you just implement the onSave method for example.*

* **An entity in removed state is no longer in persistent state.** You can check this with the contains() operation.
* **You can make the removed instance persistent again, cancelling the deletion.**
* **When the transaction commits,** Hibernate synchronizes the state transitions with the database and executes the SQL DELETE. The JVM garbage collector detects that the item is no longer referenced by anyone and finally deletes the last trace of the data.

Figure 10.4 shows the same process:



* By default, Hibernate won’t alter the identifier value of a removed entity instance. This means the item.getId() method still returns the now outdated identifier value.
* Sometimes it’s useful to work with the “deleted” data further: for example, you might want to save the removed Item again if your user decides to undo. As shown in the example, you can call persist() on a removed instance to cancel the deletion **before the persistence context is flushed.**
* Alternatively, **if you set the property** **hibernate.use\_identifier\_rollback** to **true in persistence.xml**, **Hibernate will reset the identifier value after removal of an entity instance.** In the previous code example, the identifier value is reset to the default value of null (it’s a Long). **The Item is now the same as in transient state, and you can save it again in a new persistence context.**
* Java Persistence also offers bulk operations that translate into direct SQL DELETE statements **without life cycle interceptors in the application**. We’ll discuss these operations in section 20.1.

Let’s say you load an entity instance from the database and work with the data. **For some reason, you know that another application or maybe another thread of your application has updated the underlying row in the database**. Next, we’ll see how to **refresh the data** held in memory:

## Refreshing Data

The code below shows refreshing a persistent entity instance:

Item item = em.find(Item.class, ITEM\_ID);

item.setName("Some Name");

// Someone updates this row in the database

String oldName = item.getName();

em.refresh(item);

assertNotEquals(item.getName(), oldName);

* After you load the entity instance, you realize (how isn’t important) that someone else changed the data in the database. Calling refresh() causes Hibernate to execute a SELECT to read and marshal a whole result set, overwriting changes you already made to the persistent instance in application memory.
* If the database row no longer exists (someone deleted it), Hibernate throws an EntityNotFoundException on refresh().
* Most applications don’t have to manually refresh in-memory state; concurrent modifications are typically resolved at transaction commit time.
* The best use case for refreshing is with an extended persistence context, which might span several request/response cycles and/or system transactions. While you wait for user input with an open persistence context, data gets stale, and selective refreshing may be required depending on the duration of the conversation and the dialogue between the user and the system. Refreshing can be useful to undo changes made in memory during a conversation, if the user cancels the dialogue. **We’ll have more to say about refreshing in a conversation in section 18.3.**

Another infrequently used operation is replication of an entity instance:

## Replicating Data

Replication is useful, for example, when you need to retrieve data from one database and store it in another.

An example case is a product upgrade: if the new version of your application requires a new database (schema), you may want to migrate and replicate the existing data once.

Read more about this later if you have the time to….

# Behind the Scenes of the Persistence Context, and How to Influence it

The persistence context does many things for you: **automatic dirty checking**, **guaranteed scope of object identity**, **and so on**. It’s equally important that you know some of the details of its management, and that **you sometimes influence what goes on behind the scenes:**

## Caching in the Persistence Context

The persistence context is **a cache of persistent instances**. Every entity instance in persistent state is associated with the persistence context.

Many Hibernate users who ignore this simple fact run into an **OutOfMemory-Exception.** This is typically the case when you load thousands of entity instances in a unit of work but never intend to modify them.

Hibernate still has to create a snapshot of each instance in the persistence context cache, which can lead to memory exhaustion.

(Obviously, you should execute a bulk data operation if you modify thousands of

rows—we’ll get back to this kind of unit of work in section 20.1.)

The persistence context cache never shrinks automatically. Keep the size of your persistence context to the necessary minimum.

Often, many persistent instances in your context are there by accident—for example, because you needed only a few items but queried for many. Extremely large graphs can have a serious performance impact and require significant memory for state snapshots.

Check that your queries return only data you need.

and consider the following ways to control Hibernate’s caching behavior:

* You can call EntityManager#detach(i) **to evict a persistent instance manually** from the persistence context.
* You can call EntityManager#clear() **to detach all** persistent entity instances, leaving you with an empty persistence context.

The native Session API has **some extra operations** you might find useful. You can:

* set the entire persistence context to read-only mode:
  + **This disables state snapshots and dirty checking**
  + **and Hibernate won’t write modifications to the database**
  + **you can still delete them though**
  + **Modifications to mapped collections are tricky, you should be careful**

em.unwrap(Session.class).setDefaultReadOnly(true);

Item item = em.find(Item.class, ITEM\_ID);

item.setName("New Name");

em.flush(); **---🡪 No update will be done**

You can disable dirty checking for **a single entity instance:**

Item item = em.find(Item.class, ITEM\_ID);

**em.unwrap(Session.class).setReadOnly(item, true);**

item.setName("New Name");

em.flush();**-> No UPDATE**

**A query with the org.hibernate.Query interface can return read-only results, which Hibernate doesn’t check for modifications:**

org.hibernate.Query query = em.unwrap(Session.class).createQuery("select i from Item i");

query.setReadOnly(true).list();

List<Item> result = query.list();

for (Item item : result)

item.setName("New Name");

em.flush(); **-> No UPDATE**

Thanks to query hints, you can also disable dirty checking for instances obtained with the JPA standard javax.persistence.Query interface:

Query query = em.createQuery(queryString).setHint(org.hibernate.annotations.QueryHints.READ\_ONLY,true);

* org.hibernate.annotations.QueryHints is deprecated, use AvailableHints instead

Be careful with read-only entity instances: you can still delete them, and modifications to collections are tricky! The Hibernate manual has a long list of special cases you need to read if you use these settings with mapped collections. You’ll see more query examples in chapter 14.

So far, flushing and synchronization of the persistence context have occurred automatically, when the transaction commits. In some cases, you need more control over the synchronization process:

## Flushing The Persistence Context

**By default**, Hibernate flushes the persistence context of an EntityManager and synchronizes changes with the database whenever the joined transaction is committed.

All the previous code examples, except some in the last section, have used that strategy.

JPA allows implementations to synchronize the persistence context at other times, if they wish:

Hibernate, as a JPA implementation, synchronizes at the following times:

**When a joined JTA system transaction is committed**

** Before a query is executed—we don’t mean lookup with find() but a query with javax.persistence.Query or the similar Hibernate API**

** When the application calls flush() explicitly**

You can control this behavior with the **FlushModeType** setting of an EntityManager:

tx.begin();

EntityManager em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM\_ID);

item.setName("New Name");

em.setFlushMode(FlushModeType.COMMIT);

assertEquals(

em.createQuery("select i.name from Item i where i.id = :id")

.setParameter("id", ITEM\_ID).getSingleResult(),

"Original Name"

);

tx.commit();

em.close();

Here, you load an Item instance, and change its name. Then you query the database, retrieving the item’s name.

Usually Hibernate recognizes that data has changed in memory and synchronizes these modifications with the database before the query. This is the behavior of FlushModeType.AUTO, **the default if you join the Entity-Manager with a transaction.???**

With FlushModeType.COMMIT, you’re disabling flushing before queries, so you may see different data returned by the query than what you have in memory. The synchronization then occurs only when the transaction commits.

You can at any time, while a transaction is in progress, force dirty checking and synchronization with the database by calling EntityManager#flush().

*Some notes from further testing and reading:*

* For **native queries** and how hibernate deals with them in terms of flushing behavior, take a look at the documentation. The following discussion is about JPQL, HQL queries.
* **To prevent unexpected behavior when there is a difference between hibernate and JPA, you stick with either the EntityManager API or the Session API for flushing, persisting, transactions, etc.**
* Hibernate defines FlushMode for it’s internal use instead of FlushModeType from JPA:

1. **AUTO:**

It corresponds to the AUTO flush mode from JPA and is the default mode.

I tested it on the session API, and compared to the entity manager API it works smarter and for example didn’t flush a new instance before executing **a Query that was defined by the session API: session.createQuery().**

Either way, when using JPA or hibernate this mode won’t flush every change before queries or typed queries. Sometimes you persist an instance from a table and query on another table. In this scenario the persisted entity won’t be flushed before the query cause it doesn’t need to.

**This mode is responsible to flush the changes that can affect the result of the queries.**

Hibernate will also flush changes when the transaction.commit is called

1. **COMMIT:**

Hibernate says that in this flush mode session will be flushed when transaction.commit is called and never be flushed before query executions.

but JPA says that flushing occurs at transaction commit but doesn’t specify how the implementation should flush at other times.

1. **ALWAYS:**

This is native to Hibernate and will always flush changes before query executions as opposed to the AUTO mode that will sometimes do so before query execution.

This mode will even flush for changes in a table that doesn’t have to do anything with the queried table.

This mode is usually inefficient and unnecessary

1. **MANUAL:**

This mode is also native to hibernate and will only flush the session when the flush method is called explicitly.

This is very efficient for read-only transactions.

# Working with Detached State

Next, we look at the *detached* entity state. We already mentioned some issues you’ll see when entity instances aren’t associated with a persistence context anymore, such as disabled lazy initialization. Let’s explore the detached state with some examples, so you know what to expect when you work with data outside of a persistence context.

**Did you understand why you can’t persist detached entities?**