Difference between persist/save & merge/update

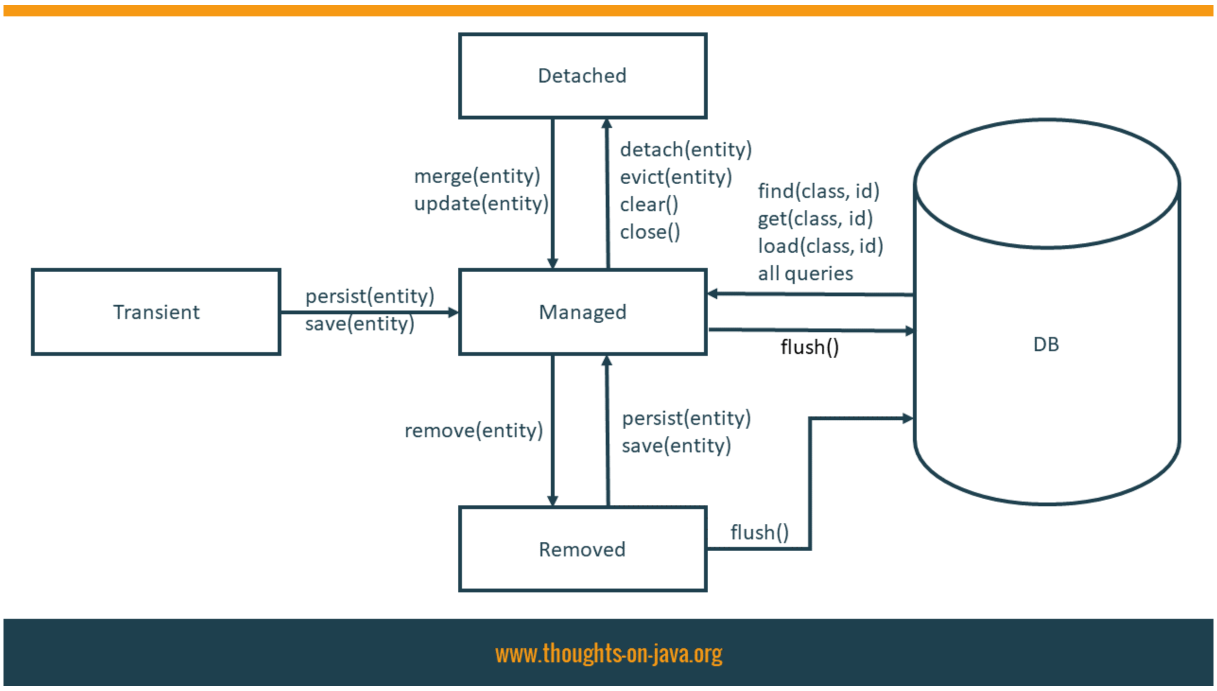
JPA and Hibernate provide different methods to persist new and to update existing entities. You can choose between JPA's persist and merge and Hibernate's save and update methods.

It seems like there are 2 pairs of 2 methods that do the same. You can use the methods persist and save to store a new entity and the methods merge and update to store the changes of a detached entity in the database.

So, which methods should you use?

Entity State Transitions

Let's take a quick look at JPA's entity lifecycle states.



Persisting A New Entity Using *persist* Or *save*

When you create a new entity object, it's in the transient lifecycle state. It does not map any database record. You need to attach the entity to a persistence context so that it becomes managed and gets persisted in the database. You can either use JPA's *persist* or Hibernate's *save* method to do that. Both methods seem to do the same, but there are a few differences.

Specification vs. Proprietary API

The most obvious difference is that the JPA specification defines the persist method. You can use it with all JPA implementations. The save method, on the other hand, is Hibernate-specific. It is, therefore, not available in other JPA implementations.

Return Types And Execution Of SQL Statements

Another obvious difference between these 2 methods is their return type. JPA's *persist* method returns *void* and Hibernate's *save* method returns the primary key of the entity.

That might seem like a huge difference, especially when you consider that Hibernate's Javadoc states that the primary key gets generated immediately while the JPA specification doesn't define when the primary key has to be generated.

In most cases, it doesn't make any difference if you call the *save* or *persist* method. Hibernate uses the name of the entity class and the primary key value to store the entity in the first level cache. It, therefore, needs a primary key value when it executes the *persist* method.

In almost all situations, Hibernate generates the primary key value immediately and triggers an SQL statement if necessary, when you call the *persist* or *save* method.

But that is not the case if you use the *IDENTITY* strategy and try to persist an entity without an active transaction or with *FlushMode.MANUAL*. If you call the persist method in one of these situations, Hibernate delays the execution of the SQL INSERT statement and creates a temporary primary key value. But if you call the *save* method, Hibernate performs the SQL INSERT statement immediately and retrieves the primary key value from the database.

Which one to choose?

You could expect that the *save* and *persist* method behave differently because there are a few differences between the JPA specification and the Javadoc of Hibernate’s proprietary methods.

But almost all of these differences disappear when you take a look at the internal implementation. The only ones that remain are 2 corner cases in which Hibernate might delay the retrieval of the primary key, the return type of the method and the support by other JPA implementations.

For most applications, it doesn’t make any difference if you get the generated primary key value as the return type of Hibernate’s *save* method or from the getter method of your primary key attribute. As long as you don’t use an extended persistence context and perform all database operations with an active transaction, I recommend using JPA’s *persist* method.

Updating a detached entity

You can use Hibernate’s *update* or JPA’s *merge* method to associate a detached entity with a persistence context. After you’ve done that, Hibernate will update the database based on the entity attribute values.

The effect of the *update* and *merge* method seem to be the same, but as you will see in the following sections, there is an important difference.

JPA’s *merge* method

JPA’s *merge* method copies the state of a detached entity to a managed instance of the same entity. Hibernate, therefore, executes an SQL SELECT statement to retrieve a managed entity from the database. If the persistence context already contained a managed instance of the entity, Hibernate uses the existing one instead. It then copies all attribute values to the managed entity and returns it to the caller.

When Hibernate flushes the persistence context for the next time, its dirty checking mechanism checks all managed entities. If it detects that the *merge* operation changed any entity attribute value, it triggers the required SQL UPDATE statement.

There is one important detail you need to know when you use JPA's *merge* method. Hibernate copies the attribute values of the detached entity to the managed entity. This overwrites any changes that you performed on this entity within the current Session.

Hibernate's *update* method

Hibernate's *update* method doesn't trigger an SQL SELECT statement. It just attaches the entity to the current persistence context. In contrast to JPA's merge method, you can't lose any changes by calling the *update* method. If the persistence context already contains a managed instance of the entity you want to update, it throws an exception.

When Hibernate performs the next flush, it doesn't perform any dirty checks. That's not possible because Hibernate didn't read the latest version of the entity from the database. It just executes an SQL UPDATE statement for the reattached entity.

The missing dirty check causes an unnecessary SQL UPDATE statement when the entity and the corresponding database record contain the same values. This might be a problem if your DBA registered an update trigger for the database table. In these situations, you can annotate your entity with *@SelectBeforeUpdate*.

That tells Hibernate to select the entity and perform a dirty check before it generates the SQL UPDATE statement. The behavior of the *update* method is now similar to JPA's *merge* method.

Which one to choose?

There is no general answer to this questions. As you have seen, both methods have their advantages and disadvantages. You have to decide for your specific use case if Hibernate needs to select the entity before it triggers the SQL UPDATE statement. And if that's the case, you also need to consider the depth of your entity graph and the performance implications of the provided fetching behavior.

Updating a managed entity

JPA and Hibernate make it very easy to update a managed entity. If your entity is in the lifecycle state managed, e.g. because you fetched it with a JPQL query or the find method of the EntityManager, you just need to change the values of your entity attributes. When Hibernate decides to flush the persistence context, the dirty checking mechanism will detect the change and perform the required SQL UPDATE statement.

You don't need to, and you should not call Hibernate's save method after you updated an entity. That triggers an additional SaveOrUpdate event without providing any benefits. When Hibernate decides to flush the persistence context, it will perform the dirty check anyway to detect all changes before it executes the required SQL UPDATE statements.

```
CREATE OR REPLACE FUNCTION calculate(IN x double
precision, IN y double precision, OUT sum double precision)
RETURNS double precision AS $BODY$ BEGIN sum = x +
y; END; $BODY$ LANGUAGE plpgsql VOLATILE COST
100;

CREATE TABLE book (id bigint NOT NULL, publishingdate
date, title character varying(255), price double precision,
version integer, CONSTRAINT book_pkey PRIMARY KEY
(id));

CREATE SEQUENCE hibernate_sequence INCREMENT 1
MINVALUE 1 MAXVALUE 9223372036854775807 START 1
CACHE 1;
```