



Hibernate: save, persist, update, merge, saveOrUpdate

Last modified: July 19, 2017

by
baeldung

Persistence

I just announced the new Spring 5 modules in REST With Spring:

[>> CHECK OUT THE COURSE](#)

1. Introduction

In this article we will discuss the differences between several methods of the *Session* interface: *save*, *persist*, *update*, *merge*, *saveOrUpdate*.

This is not an introduction to Hibernate and you should already know the basics of configuration, object-relational mapping and working with entity instances. For an introductory article to Hibernate, visit our tutorial on [Hibernate 4 with Spring](#).

2. Session as a Persistence Context Implementation

The *Session* interface has several methods that eventually result in saving data to the database: *persist*, *save*, *update*, *merge*, *saveOrUpdate*. To understand the difference between these methods, we must first discuss the purpose of the *Session* as a persistence context and the difference between the states of entity instances in relation to the *Session*.

We should also understand the history of Hibernate development that led to some partly duplicated API methods.

2.1. Managing Entity Instances

Apart from object-relational mapping itself, one of the problems that Hibernate was intended to solve is the problem of managing entities during runtime. The notion of "persistence context" is Hibernate's solution to this problem. Persistence context can be thought of as a container or a first-level cache for all the objects that you loaded or saved to a database during a session.

The session is a logical transaction, which boundaries are defined by your application's business logic. When you work with the database through a persistence context, and all of your entity instances are attached to this context, you should always have a single instance of entity for every database record that you've interacted during the session with.

In Hibernate, the persistence context is represented by *org.hibernate.Session*

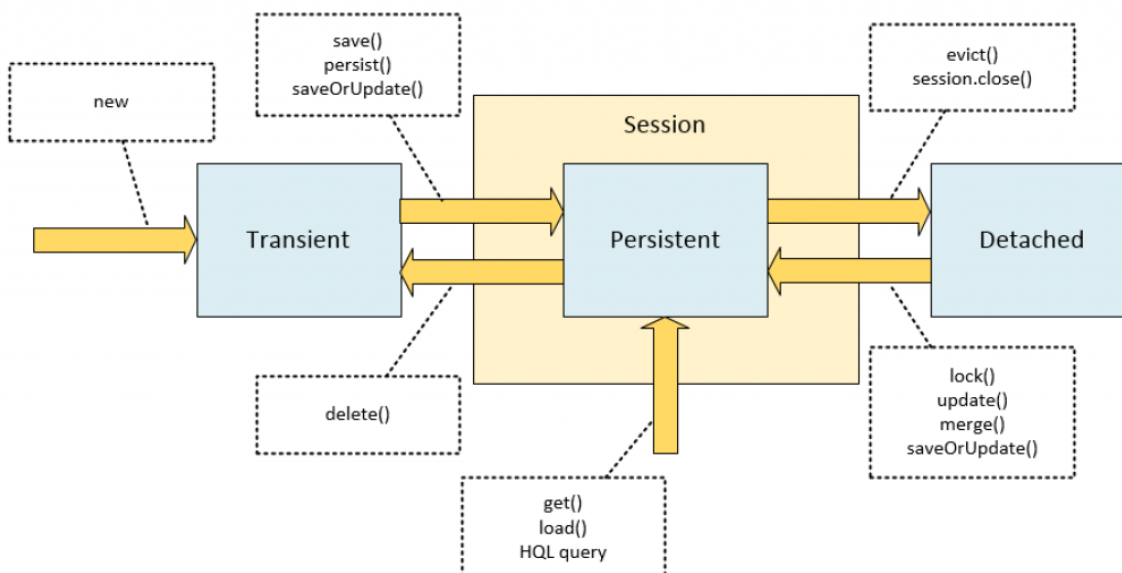
instance. For JPA, it is the *javax.persistence.EntityManager*. When we use Hibernate as a JPA provider and operate via *EntityManager* interface, the implementation of this interface basically wraps the underlying *Session* object. However, Hibernate *Session* provides a richer interface with more possibilities so sometimes it is useful to work with *Session directly*.

2.2. States of Entity Instances

Any entity instance in your application appears in one of the three main states in relation to the *Session* persistence context:

- **transient** — this instance is not, and never was, attached to a *Session*; this instance has no corresponding rows in the database; it's usually just a new object that you have created to save to the database;
- **persistent** — this instance is associated with a unique *Session* object; upon flushing the *Session* to the database, this entity is guaranteed to have a corresponding consistent record in the database;
- **detached** — this instance was once attached to a *Session* (in a **persistent** state), but now it's not; an instance enters this state if you evict it from the context, clear or close the Session, or put the instance through serialization/deserialization process.

Here is a simplified state diagram with comments on *Session* methods that make the state transitions happen.



When the entity instance is in the **persistent** state, all changes that you make to the

mapped fields of this instance will be applied to the corresponding database records and fields upon flushing the **Session**. The **persistent** instance can be thought of as "online", whereas the **detached** instance has gone "offline" and is not monitored for changes.

This means that when you change fields of a **persistent** object, you don't have to call **save**, **update** or any of those methods to get these changes to the database: all you need is to commit the transaction, or flush or close the session, when you're done with it.

2.3. Conformity to JPA Specification

Hibernate was the most successful Java ORM implementation. No wonder that the specification for Java persistence API (JPA) was heavily influenced by the Hibernate API. Unfortunately, there were also many differences: some major, some more subtle.

To act as an implementation of the JPA standard, Hibernate APIs had to be revised. Several methods were added to Session interface to match the EntityManager interface. These methods serve the same purpose as the "original" methods, but conform to the specification and thus have some differences.

3. Differences Between the Operations

It is important to understand from the beginning that all of the methods (**persist**, **save**, **update**, **merge**, **saveOrUpdate**) do not immediately result in the corresponding SQL **UPDATE** or **INSERT** statements. The actual saving of data to the database occurs on committing the transaction or flushing the **Session**.

The mentioned methods basically manage the state of entity instances by transitioning them between different states along the lifecycle.

As an example entity, we will use a simple annotation-mapped entity **Person**:

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;
```

```
// ... getters and setters

}
```

3.1. Persist

The ***persist*** method is intended for adding a new entity instance to the persistence context, i.e. transitioning an instance from transient to ***persistent*** state.

We usually call it when we want to add a record to the database (persist an entity instance):

```
Person person = new Person();
person.setName("John");
session.persist(person);
```

What happens after the ***persist*** method is called? The ***person*** object has transitioned from ***transient*** to ***persistent*** state. The object is in the persistence context now, but not yet saved to the database. The generation of ***INSERT*** statements will occur only upon committing the transaction, flushing or closing the session.

Notice that the ***persist*** method has ***void*** return type. It operates on the passed object "in place", changing its state. The ***person*** variable references the actual persisted object.

This method is a later addition to the Session interface. The main differentiating feature of this method is that it conforms to the JSR-220 specification (EJB persistence). The semantics of this method is strictly defined in the specification, which basically states, that:

- a ***transient*** instance becomes ***persistent*** (and the operation cascades to all of its relations with ***cascade=PERSIST*** or ***cascade=ALL***),
- if an instance is already ***persistent***, then this call has no effect for this particular instance (but it still cascades to its relations with ***cascade=PERSIST*** or ***cascade=ALL***),
- if an instance is ***detached***, you should expect an exception, either upon calling this method, or upon committing or flushing the session.

Notice that there is nothing here that concerns the identifier of an instance. The spec does not state that the id will be generated right away, regardless of the id generation strategy. The specification for the ***persist*** method allows the implementation to issue statements for generating id on commit or flush, and the id

is not guaranteed to be non-null after calling this method, so you should not rely upon it.

You may call this method on an already ***persistent*** instance, and nothing happens. But if you try to persist a ***detached*** instance, the implementation is bound to throw an exception. In the following example we ***persist*** the entity, ***evict*** it from the context so it becomes ***detached***, and then try to ***persist*** again. The second call to ***session.persist()*** causes an exception, so the following code will not work:

```
Person person = new Person();
person.setName("John");
session.persist(person);

session.evict(person);

session.persist(person); // PersistenceException!
```

3.2. Save

The ***save*** method is an "original" Hibernate method that does not conform to the JPA specification.

Its purpose is basically the same as ***persist***, but it has different implementation details. The documentation for this method strictly states that it persists the instance, "first assigning a generated identifier". The method is guaranteed to return the ***Serializable*** value of this identifier.

```
Person person = new Person();
person.setName("John");
Long id = (Long) session.save(person);
```

The effect of saving an already persisted instance is the same as with ***persist***. Difference comes when you try to save a ***detached*** instance:

```
Person person = new Person();
person.setName("John");
Long id1 = (Long) session.save(person);

session.evict(person);
Long id2 = (Long) session.save(person);
```

The ***id2*** variable will differ from ***id1***. The call of ***save*** on a ***detached*** instance creates a new ***persistent*** instance and assigns it a new identifier, which results in a duplicate record in a database upon committing or flushing.

3.3. Merge

The main intention of the **merge** method is to update a **persistent** entity instance with new field values from a **detached** entity instance.

For instance, suppose you have a RESTful interface with a method for retrieving an JSON-serialized object by its id to the caller and a method that receives an updated version of this object from the caller. An entity that passed through such serialization/deserialization will appear in a **detached** state.

After deserializing this entity instance, you need to get a **persistent** entity instance from a persistence context and update its fields with new values from this **detached** instance. So the **merge** method does exactly that:

- finds an entity instance by id taken from the passed object (either an existing entity instance from the persistence context is retrieved, or a new instance loaded from the database);
- copies fields from the passed object to this instance;
- returns newly updated instance.

In the following example we **evict** (detach) the saved entity from context, change the **name** field, and then **merge** the **detached** entity.

```
Person person = new Person();
person.setName("John");
session.save(person);

session.evict(person);
person.setName("Mary");

Person mergedPerson = (Person) session.merge(person);
```

Note that the **merge** method returns an object — it is the **mergedPerson** object that was loaded into persistence context and updated, not the **person** object that you passed as an argument. Those are two different objects, and the **person** object usually needs to be discarded (anyway, don't count on it being attached to persistence context).

As with **persist** method, the **merge** method is specified by JSR-220 to have certain semantics that you can rely upon:

- if the entity is **detached**, it is copied upon an existing **persistent** entity;
- if the entity is **transient**, it is copied upon a newly created **persistent** entity;
- this operation cascades for all relations with **cascade=MERGE** or **cascade=ALL** mapping;

- if the entity is **persistent**, then this method call does not have effect on it (but the cascading still takes place).

3.4. Update

As with **persist** and **save**, the **update** method is an "original" Hibernate method that was present long before the **merge** method was added. Its semantics differs in several key points:

- it acts upon passed object (its return type is **void**); the **update** method transitions the passed object from **detached** to **persistent** state;
- this method throws an exception if you pass it a **transient** entity.

In the following example we **save** the object, then **evict** (detach) it from the context, then change its **name** and call **update**. Notice that we don't put the result of the **update** operation in a separate variable, because the **update** takes place on the **person** object itself. Basically we're reattaching the existing entity instance to the persistence context — something the JPA specification does not allow us to do.

```
Person person = new Person();
person.setName("John");
session.save(person);
session.evict(person);

person.setName("Mary");
session.update(person);
```

Trying to call **update** on a **transient** instance will result in an exception. The following will not work:

```
Person person = new Person();
person.setName("John");
session.update(person); // PersistenceException!
```

3.5. SaveOrUpdate

This method appears only in the Hibernate API and does not have its standardized counterpart. Similar to **update**, it also may be used for reattaching instances.

Actually, the internal **DefaultUpdateEventListener** class that processes the **update** method is a subclass of **DefaultSaveOrUpdateListener**, just overriding some functionality. The main difference of **saveOrUpdate** method is that it does not throw

exception when applied to a *transient* instance; instead, it makes this *transient* instance *persistent*. The following code will persist a newly created instance of *Person*.

```
Person person = new Person();
person.setName("John");
session.saveOrUpdate(person);
```

You may think of this method as a universal tool for making an object *persistent* regardless of its state whether it is *transient* or *detached*.

4. What to Use?

If you don't have any special requirements, as a rule of thumb, you should stick to the *persist* and *merge* methods, because they are standardized and guaranteed to conform to the JPA specification.

They are also portable in case you decide to switch to another persistence provider, but they may sometimes appear not so useful as the "original" Hibernate methods, *save*, *update* and *saveOrUpdate*.

5. Conclusion


We've discussed the purpose of different Hibernate Session methods in relation to managing persistent entities in runtime. We've learned how these methods transit entity instances through their lifecycles and why some of these methods have duplicated functionality.

The source code for the article is [available on GitHub](#).

I just announced the new Spring 5 modules in REST With Spring:

>> CHECK OUT THE LESSONS

☐ Subscribe
 ☐ newest
 ☐ oldest
 ☐ most voted



lewy


☐
☐

If merge operation can save transient object what is the purpose of using persist operation?

☐ 0 ☐

☐ 1 year ago

☐



Sergey Petunin

☐
☐

The merge operation does not make the passed object persistent, it always copies passed object onto another object in the persistence context. You actually can use only merge operation in your application, but that's an important architecture decision: you'll always have to discard the object passed to the merge operation.

This means that if some components of your application hold on to this object, their reference has to be updated. So choosing between merge-only or persist+merge is a matter of architectural style.

☐ 2 ☐

☐ 1 year ago




alexey semenyuk

☐
☐

You use persist when you would like that the method creates a new entity and never updates an entity. In otherwise the method throws an exception about primary key uniqueness violation. In this case you handle entities in a stateful manner and can use gateway pattern. You use merge if you would like that the method either inserts or updates an entity in the database. In this case you handle entities in a stateless manner and can use data transfer objects in services.

☐ 1 ☐

☐ 1 year ago



funhoabin

☐
☐

when does merge operation decide to save a object as new? so i think that first the merge operation will find a object in persist context if that hasn't one it will find by query to sql to get it. if it not has too. merge operation will make object as a newly persist object??

1

1 year ago



Grzegorz Piwowarek



If you jump to JPA specification, you will see there:

"The semantics of the merge operation applied to an entity X are as follows:

If X is a detached entity, the state of X is copied onto a pre-existing managed entity instance X' of the same identity or a new managed copy X' of X is created.

If X is a new entity instance, a new managed entity instance X' is created and the state of X is copied into the new managed entity instance X'."

So, the answer is "yes". Cheers!

0

1 year ago



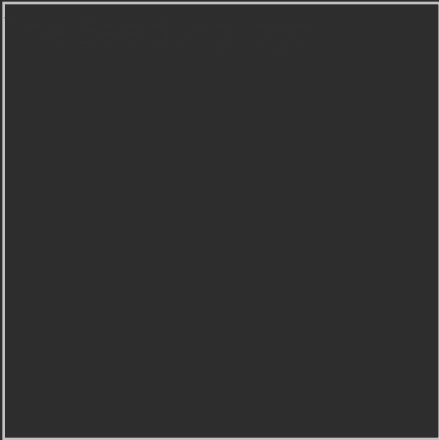
Vusala



Thank you very much!

0

1 year ago



CATEGORIES

SPRING
REST
JAVA
SECURITY
PERSISTENCE
JACKSON
HTTPCLIENT
KOTLIN

SERIES

JAVA "BACK TO BASICS"
TUTORIAL
JACKSON JSON TUTORIAL
HTTPCLIENT 4 TUTORIAL
REST WITH SPRING TUTORIAL
SPRING PERSISTENCE
TUTORIAL
SECURITY WITH SPRING

ABOUT

ABOUT BAELDUNG
THE COURSES
CONSULTING WORK
META BAELDUNG
THE FULL ARCHIVE
WRITE FOR BAELDUNG
CONTACT
COMPANY INFO
TERMS OF SERVICE
PRIVACY POLICY
EDITORS
MEDIA KIT (PDF)

