(/)

JPA Entity Lifecycle Events

Last modified: July 29, 2020

by Amy DeGregorio (https://www.baeldung.com/author/amy-degregorio/)

Persistence (https://www.baeldung.com/category/persistence/)

JPA (https://www.baeldung.com/tag/jpa/)

1. Introduction

When working with JPA, there are several events that we can be notified of during an entity's lifecycle. **In this tutorial, we'll discuss the JPA entity lifecycle events and how we can use annotations to handle the callbacks and execute code when these events occur.**

We'll start by annotating methods on the entity itself and then move on to using an entity listener.

2. JPA Entity Lifecycle Events

JPA specifies seven optional lifecycle events that are called:

- before persist is called for a new entity – *@PrePersist*
- after persist is called for a new entity – *@PostPersist*
- before an entity is removed – *@PreRemove*
- after an entity has been deleted – *@PostRemove*
- before the update operation – *@PreUpdate*
- after an entity is updated – *@PostUpdate*
- after an entity has been loaded – *@PostLoad*

There are two approaches for using the lifecycle event annotations: annotating methods in the entity and creating an *EntityListener* with annotated callback methods. We can also use both at the same time. Regardless of where they are, **callback methods are required to have a *void* return type.**

So, if we create a new entity and call the *save* method of our repository, our method annotated with *@PrePersist* is called, then the record is inserted into the database, and finally, our *@PostPersist* method is called. **If we're using *@GeneratedValue* to automatically generate our primary keys, we can expect that key to be available in the *@PostPersist* method.**

For the *@PostPersist*, *@PostRemove* and *@PostUpdate* operations, the documentation mentions that these events can happen right after the operation occurs, after a flush, or at the end of a transaction.

We should note that the *@PreUpdate* callback is only called if the data is actually changed — that is if there's an actual SQL update statement to run. The *@PostUpdate* callback is called regardless of whether anything actually changed.

If any of our callbacks for persisting or removing an entity throw an exception, the transaction will be rolled back.

3. Annotating the *Entity*

Let's start by using the callback annotations directly in our entity. In our example, we're going to leave a log trail when *User* records are changed, so we're going to add simple logging statements in our callback methods.

Additionally, we want to make sure we assemble the user's full name after they're loaded from the database. We'll do that by annotating a method with *@PostLoad*.

We'll start by defining our *User* entity:

```
@Entity
public class User {
    private static Log log = LogFactory.getLog(User.class);

    @Id
    @GeneratedValue
    private int id;

    private String userName;
    private String firstName;
    private String lastName;
    @Transient
    private String fullName;

    // Standard getters/setters
}
```

Next, we need to create a *UserRepository* interface:

```
public interface UserRepository extends JpaRepository<User, Integer> {
    public User findByUserName(String userName);
}
```

Now, let's return to our *User* class and add our callback methods:

```
@PrePersist
public void logNewUserAttempt() {
    log.info("Attempting to add new user with username: " + userName);
}

@PostPersist
public void logNewUserAdded() {
    log.info("Added user '" + userName + "' with ID: " + id);
}

@PreRemove
public void logUserRemovalAttempt() {
    log.info("Attempting to delete user: " + userName);
}

@PostRemove
public void logUserRemoval() {
    log.info("Deleted user: " + userName);
}

@PreUpdate
public void logUserUpdateAttempt() {
    log.info("Attempting to update user: " + userName);
}

@PostUpdate
public void logUserUpdate() {
    log.info("Updated user: " + userName);
}

@PostLoad
public void logUserLoad() {
    fullName = firstName + " " + lastName;
}
```

When we run our tests, we'll see a series of logging statements coming from our annotated methods. Additionally, we can reliably expect our user's full name to be populated when we load a user from the database.

4. Annotating an *EntityListener*

We're going to expand on our example now and use a separate *EntityListener* to handle our update callbacks. **We might favor this approach over placing the methods in our entity if we have some operation we want to apply to all of our entities.**

Let's create our *AuditTrailListener* to log all the activity on the *User* table:

```
public class AuditTrailListener {
    private static Log log =
LogFactory.getLog(AuditTrailListener.class);

    @PrePersist
    @PreUpdate
    @PreRemove
    private void beforeAnyUpdate(User user) {
        if (user.getId() == 0) {
            log.info("[USER AUDIT] About to add a user");
        } else {
            log.info("[USER AUDIT] About to update/delete user: " +
user.getId());
        }
    }

    @PostPersist
    @PostUpdate
    @PostRemove
    private void afterAnyUpdate(User user) {
        log.info("[USER AUDIT] add/update/delete complete for user: " +
user.getId());
    }

    @PostLoad
    private void afterLoad(User user) {
        log.info("[USER AUDIT] user loaded from database: " +
user.getId());
    }
}
```

As we can see from the example, **we can apply multiple annotations to a method**.

Now, we need to go back to our *User* entity and add the *@EntityListener* annotation to the class:

```
@EntityListeners(AuditTrailListener.class)
@Entity
public class User {
    //...
}
```

And, when we run our tests, we'll get two sets of log messages for each update action and a log message after a user is loaded from the database.

5. Conclusion

In this article, we've learned what the JPA entity lifecycle callbacks are and when they're called. We looked at the annotations and talked about the rules for using them. We've also experimented with using them in both an entity class and with an *EntityListener* class.

The example code is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-jpa-annotations).