**JPA / Hibernate – Mapping One-to-One relationships[[1]](#footnote-1)**

* [Home](https://nullbeans.com/)
* [java](https://nullbeans.com/category/java/)
* JPA / Hibernate – Mapping One-to-One relationships

Posted in [java](https://nullbeans.com/category/java/), [JPA](https://nullbeans.com/category/jpa/), [Spring](https://nullbeans.com/category/spring/) By [iba](https://nullbeans.com/author/doctormohsen/)Posted on [February 25, 2019](https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/)

In this tutorial, we will discuss how to map and configure One-to-One relationships in JPA and Hibernate.

**Contents** [hide](https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/)

[1 What are One-To-One relationships?](https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/#What_are_One-To-One_relationships)

[2 Configuring One-To-One relationships](https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/#Configuring_One-To-One_relationships)

[3 Lazy vs Eager FetchType for One-To-One relationship](https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/#Lazy_vs_Eager_FetchType_for_One-To-One_relationship)

[4 Summary](https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/#Summary)

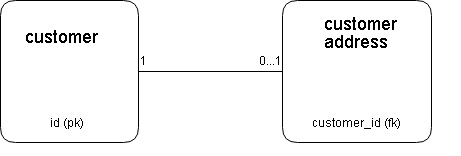
[5 Related Posts](https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/#Related_Posts)

**What are One-To-One relationships?**

One to One relationships in databases define the relationship between two different entities, where the first entity can be related to zero or one of the second entity. For example, a user can have one profile picture configured on their profile page (or not).

**Configuring One-To-One relationships**

In our example, we will be mapping the relationship between a Customer entity and a CustomerAddress entity. The following graphic provides an overview of the relationship:



Note that the customer\_address database table holds the customer\_id foreign key. Therefore, we will chose the customer\_address as the **owning side** of the relationship. Configuring One-To-One relationships in JPA is relatively simple compared to One-To-Many and Many-To-Many relationships. First, we start by mapping the CustomerAddress class as follows:

**CustomerAddress.java**

|  |
| --- |
| @Entity @Table(name = "**customer\_address**")  public class **CustomerAddress** {  private long **id**;  private Customer **customer**;  .......... .......... .......... ..........  @**OneToOne**(optional = **false**)  @JoinColumn(name = "**customer\_id**", referencedColumnName = "**id**")  public Customer getCustomer() { return customer; }  public void setCustomer(Customer customer) { this.customer = customer; } } |

Let us quickly go through the annotations that were used:

* **@OneToOne**:

This indicates that the property maps a One-To-One relationship.

The optional = false indicates that the “**Customer**” property cannot be null.

* **@JoinColumn**:

Since the foreign key exists in the customer\_address table, then we ask Hibernate to map our foreign key id to the id field of the customer entity.

Usually Hibernate is smart enough to deduce the name of the table and the primary key fiend of the referenced entity (in this case, the Customer entity), therefore the **referencedColumnName**can be omitted.

We can leave the configuration as it is. But it is currently inconvenient to edit the customer’s address via the Customer entity as we do not have a reference from the Customer entity to the corresponding CustomerAddress. Therefore, we will need to configure the relationship **bidirectionally**. Let us start with the Customer entity class:

|  |
| --- |
| @Entity  @Table(name = "**customer**")  public class **Customer** {  private long id;  private CustomerAddress customerAddress;  ........ ........  @**OneToOne**(**mappedBy = "customer", cascade = CascadeType.ALL**)  public CustomerAddress getCustomerAddress() { return customerAddress; }  public void setCustomerAddress(CustomerAddress customerAddress) {  this.customerAddress = customerAddress; } } |

In the Customer class, we used the @OneToOne annotation again. But this time, we set the **mappedBy** property. The mappedBy property indicates the property or field on the **other**side of the relationship that maps to the current entity. In other words, the property in the CustomerAddress class that maps to the Customer class is “customer”. This is important to indicate as it can happen that multiple One-to-One relationships exist between the same entity classes. For example, one instance of a CustomerAddress can be used to indicate a home address and the other one can be used to indicate a work address.

We also used the **CascadeType.ALL** as we would like to save the customer address when we save the customer as well. We would also like to delete the customer’s address when we delete the customer. Therefore, cascading all events is desired in this case.

The final piece of the configuration is to **configure the bidirectional setter on the owning side of the relationship.**A bidirectional setter (as the name implies) sets both sides of the relationship. This is crucial in order to avoid data inconsistencies in memory and to avoid transient entity exceptions. Please check our guide [here](https://nullbeans.com/dataintegrityviolationexception-constraintviolationexception-managing-jpa-bidirectional-relationships/) to learn more about data consistency in memory. Since the CustomerAddess is the owning side of the relationship in our example, we will modify the setCustomer method as follows:

|  |
| --- |
| /\*\*  \* Bidirectional setter  \* @param customer  \*/  public void **setCustomer**(Customer customer) {  this.customer = customer;    if(customer!=null) {  customer.setCustomerAddress(this);  }  } |

When we create a new instance of the CustomerAddress entity, we need to set the customer through the bidirectional setter of the CustomerAddess entity as it is the owning side of the relationship. This also sets the customer’s side of the relationship, therefore insuring consistency of the data. The null check exists to protect against NullPointerExceptions.

**Lazy vs Eager FetchType for One-To-One relationship**

**By default, One-To-One relationships are fetched eagerly** by Hibernate. However, the choice of whether to do a lazy fetch or not depends highly on the application. For example, if you expect that you will need the customer address whenever you are handling a customer, then it would be better to use eager loading, as it would save you a round trip to the database or the hibernate cache. However, if you have a bulky entity at the other side of the relationship which will be needed only in special occasions, then it is better to optimize your database access performance by using lazy loading.

If you would like to configure the customer to lazyily fetch the customer address, then you can do it as follows:

|  |
| --- |
| @**OneToOne**(mappedBy = "**customer**", cascade = CascadeType.ALL, **fetch = FetchType.LAZY**) public CustomerAddress getCustomerAddress() {    return customerAddress;  } |

**Summary**

In this article, we discussed what is a One-To-One relationship and how to configure it in JPA. We discussed why we would need to map the relationship in a bidirectional manner and how to do it in a way to insure consistency of the data in memory. We later discussed if we should perform lazy loading or not in One-To-One relationships.

**Related Posts**

* [Configuring and mapping database tables to JPA entities](https://nullbeans.com/configuring-and-mapping-database-tables-to-jpa-entity-classes/)
* [How to manage JPA bidirectional relationships properly](https://nullbeans.com/dataintegrityviolationexception-constraintviolationexception-managing-jpa-bidirectional-relationships/)
* [Configuring @PostLoad and @PostUpdate in JPA/Hibernate](https://nullbeans.com/configuring-postload-and-postupdate-in-jpa-hibernate/)
* [JPA / Hibernate troubleshooting](https://nullbeans.com/jpa-hibernate-troubleshooting-guides/)
* [How to use @PrePersist and @PreUpdate in JPA/Hibernate](https://nullbeans.com/how-to-use-prepersist-and-preupdate-in-jpa-hibernate/)
* [org.hibernate.MappingException: Could not determine…](https://nullbeans.com/jpa-hibernate-accesstype-mappingexception/)

1. https://nullbeans.com/jpa-hibernate-mapping-one-to-one-relationships/ [↑](#footnote-ref-1)