[General Questions 3](#_Toc68777493)

[1) What is JPA? 3](#_Toc68777494)

[2) What is object relation mapping? 3](#_Toc68777495)

[3) Advantages of JPA? 3](#_Toc68777496)

[4) List some ORM frameworks? 4](#_Toc68777497)

[5) Explain JPA Architecture? 4](#_Toc68777498)

[6) Different module of JPA 7](#_Toc68777499)

[7) Difference between JPA, Hibernate, Spring data JPA 7](#_Toc68777500)

[8) Current version 7](#_Toc68777501)

[9) Persist vs Merge method 8](#_Toc68777502)

[10) Is it possible to use JPA with noSQl databases? 8](#_Toc68777503)

[11) What is the difference between JPA and JDO? 8](#_Toc68777504)

[12) Can the Entity class inherit from non-Entity classes? 9](#_Toc68777505)

[13) Can an Entity class inherit from other Entity classes? 9](#_Toc68777506)

[14) Can a non-Entity class inherit from an Entity class? 9](#_Toc68777507)

[15) What data types are allowed in the attributes of the Entity class (fields or properties)? 9](#_Toc68777508)

[16) What data types can be used in the attributes included in the primary key of the Entity class (composite or simple) so that the resulting primary key can be used for any database? And in the case of auto-generated primary key (generated primary keys)? 10](#_Toc68777509)

[17) Properties of entity? 10](#_Toc68777510)

[18) How can I get JPA metadata (information about Entity types, Embeddable and Managed classes, etc.)? 10](#_Toc68777511)

[19) ​​What is meant by polymorphism (polymorphism) in JPQL queries (Java Persistence query language) and how to “turn it off”? 11](#_Toc68777512)

[20) What is the difference in Hibernate Entity requirements from the Entity requirements in the JPA specification (see question 10)? 11](#_Toc68777513)

[21) What is the unique inheritance strategy in Hibernate, but not in the JPA specification? 11](#_Toc68777514)

[22) What are the main new features in the JPA 2.1 specification compared with JPA 2.0 (list at least five or six new features)? 11](#_Toc68777515)

[23) What annotations can overlap connections (override entity relationship) or attributes inherited from the superclass, or specified in the embeddable class when using this embeddable class in one of the entity classes and do not overlap in the others? 12](#_Toc68777516)

[Annotations meaning 13](#_Toc68777517)

[24) @Transient notation in JPA? 13](#_Toc68777518)

[25) @JoinTable 13](#_Toc68777519)

[26) @Basic 16](#_Toc68777520)

[27) @Column and difference between @Basic and @ Column?? 17](#_Toc68777521)

[JPA Sample Example 17](#_Toc68777522)

[CORE of JPA 17](#_Toc68777523)

[28) Entities: 17](#_Toc68777524)

[29) Access Strategy in JPA 18](#_Toc68777525)

[30) How to override default strategy and can we mix both strategy in one entity? 19](#_Toc68777526)

[31) Which access strategy we should use? 19](#_Toc68777527)

[32) How to define Primary key and how to create composite Primary key? 19](#_Toc68777528)

[33) How to define composite Primary key? 20](#_Toc68777529)

[34) @IDClass annotation 20](#_Toc68777530)

[35) @EmbeddedId 21](#_Toc68777531)

[36) @IdClass vs @EmbeddedId 22](#_Toc68777532)

[37) What are the requirements for Primary Key class? 23](#_Toc68777533)

[38) What is the Embeddable class? 23](#_Toc68777534)

[39) Can the Embeddable class contain another Embeddable class? 23](#_Toc68777535)

[40) Can the Embeddable class contain relationships with other Entity or Entity collections? If it can, are there any restrictions on such relationships? 24](#_Toc68777536)

[41) What requirements does JPA set for Embeddable classes? 24](#_Toc68777537)

[42) How to persist entity to databases? 24](#_Toc68777538)

[43) Entity manager methods to find, delete, insert and update? 25](#_Toc68777539)

[44) Direction in entities 26](#_Toc68777540)

[45) Cascade Operations and Relationships 27](#_Toc68777541)

[46) Multiplicity Entity Relationships – One to one 28](#_Toc68777542)

[47) OnetoOne unidirectional example. 30](#_Toc68777543)

[48) One to Many / Many to one 32](#_Toc68777544)

[49) Many to many 33](#_Toc68777545)

[50) Orphan removal 33](#_Toc68777546)

[51) Difference between orphanRemoval and cascade.remove? 34](#_Toc68777547)

[52) Inheritance in entity 34](#_Toc68777548)

[53) Annotations used in Entity? 34](#_Toc68777549)

[54) Collection in JPA? 36](#_Toc68777550)

[55) Difference between @OneToMany and @ElementCollection 38](#_Toc68777551)

[56) Id generation types in JPA 39](#_Toc68777552)

[57) Entity LifeCycle 41](#_Toc68777553)

[58) Entity callback methods and event listener 42](#_Toc68777554)

[59) Type of lock in JPA?? 43](#_Toc68777555)

[60) How can you change the fetch strategy settings of any Entity attributes for individual queries (query) or search methods (find), then if Enity has an attribute with fetchType = LAZY, but for a specific query you need to make it EAGER or vice versa? 43](#_Toc68777556)

[61) @Transactional 43](#_Toc68777557)

[JPQL 44](#_Toc68777558)

[62) What is JPQL and why should we use this? 44](#_Toc68777559)

[CRITERIA API 44](#_Toc68777560)

[63) What is criteria api? 44](#_Toc68777561)

[Cache 44](#_Toc68777562)

[64) Types of cache in JPa? 45](#_Toc68777563)

[65) How to enable second level cache in hibernate? 45](#_Toc68777564)

[66) What are the options for setting the second-level cache (second-level cache) in JPA or what similarly describe what values ​​can the shared-cache-mode element take from persistence.xml? 46](#_Toc68777565)

[67) Cache API to delete object from cache? 47](#_Toc68777566)

[68) How to make entity cacheable? 47](#_Toc68777567)

[Hibernate only features 47](#_Toc68777568)

[REFERENCES 47](#_Toc68777569)

# General Questions

## What is JPA?

### The Java Persistence API (JPA) is the specification of Java that is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems. As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. Therefore, ORM tools like Hibernate, TopLink, and iBatis implements JPA specifications for data persistence. The first version of the Java Persistence API, JPA 1.0 was released in 2006 as a part of EJB 3.0 specification.

## What is object relation mapping?

The object-relational mapping is a mechanism which is used to develop and maintain a relationship between an object and the relational database by mapping an object state into the database column. It converts attributes of programming code into columns of the table. It is capable of handling various database operations easily such as insertion, updation, deletion, etc.



<https://www.tutorialspoint.com/jpa/jpa_orm_components.htm>

## Advantages of JPA?

The advantages of JPA are given below.

* The burden of interacting with the database reduces significantly by using JPA.
* The user programming becomes easy by concealing the O/R mapping and database access processing.
* The cost of creating the definition file is reduced by using annotations.
* We can merge the applications used with other JPA providers
* Using different implementations can add the features to the standard Implementation which can later be the part of JPA specification.

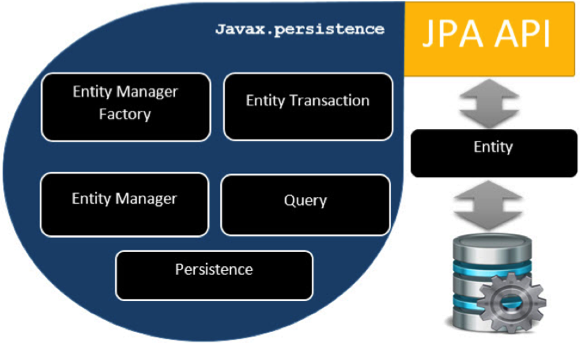
## List some ORM frameworks?

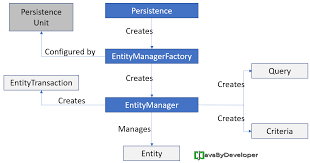
Following are the various frameworks that function on ORM mechanism: -

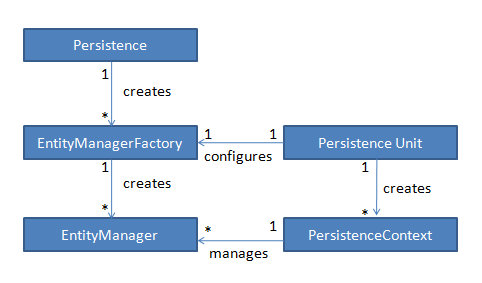
* Hibernate
* TopLink
* ORMLite
* iBATIS
* JPOX
* Eclipse Link

## Explain JPA Architecture?

Class level architecture:







### Let's describe each of the units shown in the above architecture.

### [**EntityManagerFactory**](http://www.javaguides.net/2018/12/jpa-entitymanagerfactory-interface-with-example.html) — This is a factory class of [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank). It creates and manages multiple EntityManager instances.

### [**EntityManager**](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html) — It is an interface; it manages the persistence operations on objects. It works like a factory for Query instance.

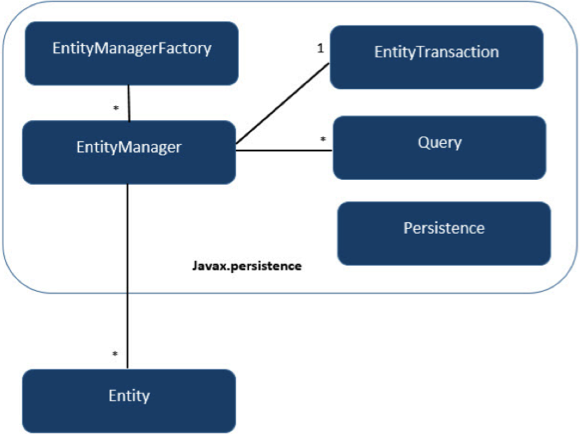
### **Entity** — Entities are the persistence objects, stored as records in the database.

### [**EntityTransaction**](http://www.javaguides.net/2018/12/jpa-entitytransaction-interface-with-example.html) — It has one-to-one relationship with [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank). For each EntityManager, operations are maintained by the [EntityTransaction](http://www.javaguides.net/2018/12/jpa-entitytransaction-interface-with-example.html" \t "_blank) class.

### **Persistence** — This class contains static methods to obtain the EntityManagerFactory instance.

### **Query** - This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.

### **JPA Class Relationships**

In the above architecture, the relations between the classes and interfaces belong to the javax.persistencepackage. The following diagram shows the relationship between them.[](https://4.bp.blogspot.com/-fVjv79bHZMA/XAPvFu2YTDI/AAAAAAAAFE4/iaL19yzi2boDDhVTffQWInYFlNVwYDDOQCLcBGAs/s1600/jpa_class_relationships.png)

### The relationship between [EntityManagerFactory](http://www.javaguides.net/2018/12/jpa-entitymanagerfactory-interface-with-example.html" \t "_blank) and [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank) is one-to-many. It is a factory class to [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank) instances.

### The relationship between [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank) and EntityTransaction is one-to-one. For each EntityManager operation, there is an EntityTransaction instance.

### The relationship between [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank) and Query is one-to-many. Many numbers of queries can execute using one [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank) instance.

### The relationship between [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank) and Entity is one-to-many. One [EntityManager](http://www.javaguides.net/2018/12/jpa-entitymanager-interface-with-example.html" \t "_blank) instance can manage multiple Entities.

## Different module of JPA

* Core – entities and relationship
* Criteria
* JPQL
* Native Query
* Named Query
* Transactions

## Difference between JPA, Hibernate, Spring data JPA

### JPA is specification. It alone does not perform any operation. Hibernate is its implementation. There are other implementation as well.

Spring data JPA is a spring abstraction layer aimed to develop application easily.

## Current version

JPA- 2.3

## Persist vs Merge method

### Both persist and merge serve different purposes.

|  |  |
| --- | --- |
| Persist | Merge |
| Persist takes an entity instance, adds it to the context and makes that instance managed | Merge creates a new instance of your entity, copies the state from the supplied entity, and makes the new copy managed. |
| Insert a new register to the database. | Find an attached object with the same id and update it. |
| You want the method always creates a new entity and never updates an entity. Otherwise, the method throws an exception as a consequence of primary key uniqueness violation. | You want the method either inserts or updates an entity in the database. |

## Is it possible to use JPA with noSQl databases?

### In general, the JPA specification says only about mapping java objects into relational database tables, but there are a number of implementations of this standard for noSql databases: Kundera, DataNucleus, ObjectDB, and a number of others. Naturally, not all specification-specific features for relational databases are transferred to nosql databases completely.

## What is the difference between JPA and JDO?

### JPA (Java Persistence API) and Java Data Objects (JDO) are two specifications for storing java objects in databases. If JPA is concentrated only on relational databases, then JDO is a more general specification that describes the ORM for any possible bases and repositories.

### In principle, JPA can be viewed as part of the JDO specification specialized in relational databases, even though the API of these two specifications does not completely match. The “developers” of specifications also differ – if JPA is developed as JSR, then JDO was first developed as JSR, now it is developed as an Apache JDO project.

## Can the Entity class inherit from non-Entity classes?

Can.

## Can an Entity class inherit from other Entity classes?

### The same can.

## Can a non-Entity class inherit from an Entity class?

### And this is also permissible.

## What data types are allowed in the attributes of the Entity class (fields or properties)?

### Valid attribute types for Entity classes are:

### primitive types and their Java wrappers,

### strings,

### any Java serializable types (implementing the Serializable interface),

### enums;

### entity types;

### embeddable classes

### and collection types 1-6

## What data types can be used in the attributes included in the primary key of the Entity class (composite or simple) so that the resulting primary key can be used for any database? And in the case of auto-generated primary key (generated primary keys)?

### Valid attribute types included in the primary key are:

### primitive types and their Java wrappers,

### strings,

### BigDecimal and BigInteger,

### java.util.Date and java.sql.Date

### In the case of an auto-generated primary key (generated primary keys), only numeric types are allowed.

### If other types of data are used in the primary key, it can work only for some databases, i.e. becomes not portable (not portable).

## Properties of entity?

### **Persistability:** An object is called persistent if it is stored in the database and can be accessed anytime.

### **Persistent Identity:** In Java, each entity is unique and represents an object identity. Similarly, when the object identity is stored in a database, then it is represented as persistence identity. This object identity is equivalent to the primary key in the database.

### **Transactionality:** A transaction is a set of operations that either fail or succeed as a unit. Transactions are a fundamental part of persistence.

### **Granularity:** Entities should not be primitives, primitive wrappers or built-in objects with single dimensional state.

<https://codingcompiler.com/jpa-interview-questions-answers/>

<https://www.javacodemonk.com/hibernate-and-spring-data-jpa-interview-questions-370a08b2>

## How can I get JPA metadata (information about Entity types, Embeddable and Managed classes, etc.)?

### JPA uses the Metamodel interface to get this information. The object of this interface can be obtained by the getMetamodel method from an EntityManagerFactory or EntityManager.

## ​​What is meant by polymorphism (polymorphism) in JPQL queries (Java Persistence query language) and how to “turn it off”?

Unlike SQL, JPQL queries have automatic polymorphism, that is, each Entity request returns not only the objects of this Entity, but also objects of all its descendant classes, regardless of the inheritance strategy (for example, the select \* from Animal query will return not only Animal objects, but also objects of Cat and Dog classes that are inherited from Animal). To eliminate this behavior, the TYPE function is used in the where condition (for example, select a from Animal a where TYPE (a) IN (Animal, Cat) will not return objects of the class Dog

## What is the difference in Hibernate Entity requirements from the Entity requirements in the JPA specification (see question 10)?

### 1) A constructor with no arguments in Hibernate is not required to be public or protected, it is recommended that it be at least a package of visibility, but this is only a recommendation, if the Java security settings allow access to private fields, then it can be private.

### 2) JPA categorically requires not to use final classes, Hibernate only recommends not using such classes so that it can create a proxy for lazy loading, but it allows you to either turn off the Proxy proxy (lazy = false), or use an interface containing all mapping methods of this class (proxy annotation (proxyClass = interface.class))

## What is the unique inheritance strategy in Hibernate, but not in the JPA specification?

### Unlike JPA, Hibernate has a unique inheritance strategy called implicit polymorphism.

## What are the main new features in the JPA 2.1 specification compared with JPA 2.0 (list at least five or six new features)?

### In the JPA 2.1 specification, there are:

### 1) Entity Graphs – a dynamic change fetchType mechanism for each request,

### 2) Converters – a mechanism for defining converters for specifying functions for converting Entity attributes into database fields,

### 3) DDL generation – automatic generation of tables, indexes and schemas,

### 4) Stored Procedures – a mechanism for calling stored procedures from JPA,

### 5) Criteria Update / Delete – a mechanism for invoking bulk updates or deletes using the Criteria API,

### 6) Unsynchronized persistence contexts – an opportunity to specify SynchronizationType,

### 7) New features in JPQL / Criteria API : arithmetic subqueries, generic database functions, join ON clause, TREAT function,

### 8) Dynamic creation of named queries (Details queries) Learn more about changing interfaces and APIs in JPA 2.1:

### 1) The EntityManager interface received new methods createStoredProcedureQuery, isJoinedToTransaction and createQuery (CriteriaUpdate or CriteriaDelete)

### 2) The Abstract AbstractQuery class inherited from the CommonAbritCritery class interfaces CriteriaUpdate, CriteriaDelete inherited CommonAbstractCriteria,

### 3) PersistenceProvider got new features generateSchema possible to generate diagrams,

### 4) received EntityManagerFactory methods addNamedQuery, unwrap, addNamedEntityGraph, createEntityManager (indicating SynchronizationType)

### 5) a new enum SynchronizationType, Entity Graphs, StoredProced ureQuery and AttributeConverter interfaces.

## What annotations can overlap connections (override entity relationship) or attributes inherited from the superclass, or specified in the embeddable class when using this embeddable class in one of the entity classes and do not overlap in the others?

### There are four annotations for this overlap:

### AttributeOverride to overlap fields, properties and primary keys,

### AttributeOverrides can similarly

### override fields, properties and primary keys with multiple values, 3. AssociationOverride to override entity relationship,

### AssociationOverrides to overlap multiple relationships.

# Annotations meaning

## @Transient notation in JPA?

### Java's transient keyword is used to denote that a field is not to be serialized, whereas JPA's @Transient annotation is used to indicate that a field is not to be persisted in the database, i.e. their semantics are different. Is it possible to set a default value for columns in JPA, and if, how is it done using annotations? Actually it is possible in JPA, although a little bit of a hack using the columnDefinition property of the @Column annotation, for example: @Column(name="Price", columnDefinition="Decimal(10,2) default '100.00'")

## @JoinTable

### When Mapping table is used between two tables to map the relationships. Typical exam is Many to many. Means when an intermediator table is required to map the relationship between 2 tables.

### <https://stackoverflow.com/questions/5478328/in-which-case-do-you-use-the-jpa-jointable-annotation>

### the JoinTable example does not need the mappedBy annotation attribute. In fact, recent versions of Hibernate refuse to start up by printing the following error:

### org.hibernate.AnnotationException:

### Associations marked as mappedBy must not define database mappings

### like @JoinTable or @JoinColumn

### Let's pretend that you have an entity named Project and another entity named Task and each project can have many tasks.

### You can design the database schema for this scenario in two ways.

### The first solution is to create a table named Project and another table named Task and add a foreign key column to the task table named project\_id:

### Project Task

### ------- ----

### id id

### name name

### project\_id

### This way, it will be possible to determine the project for each row in the task table. If you use this approach, in your entity classes you won't need a join table:

@Entity

public class Project {

@OneToMany(mappedBy = "project")

private Collection<Task> tasks;

}

@Entity

public class Task {

@ManyToOne

private Project project;

}

The other solution is to use a third table, e.g. Project\_Tasks, and store the relationship between projects and tasks in that table:

Project Task Project\_Tasks

------- ---- -------------

id id project\_id

name name task\_id

The Project\_Tasks table is called a "Join Table". To implement this second solution in JPA you need to use the @JoinTable annotation. For example, in order to implement a uni-directional one-to-many association, we can define our entities as such:

**Project entity:**

@Entity

public class Project {

@Id

@GeneratedValue

private Long pid;

private String name;

@JoinTable

@OneToMany

private List<Task> tasks;

public Long getPid() {

return pid;

}

public void setPid(Long pid) {

this.pid = pid;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public List<Task> getTasks() {

return tasks;

}

public void setTasks(List<Task> tasks) {

this.tasks = tasks;

}

}

**Task entity:**

@Entity

public class Task {

@Id

@GeneratedValue

private Long tid;

private String name;

public Long getTid() {

return tid;

}

public void setTid(Long tid) {

this.tid = tid;

}

public String getName() {

return name;

}

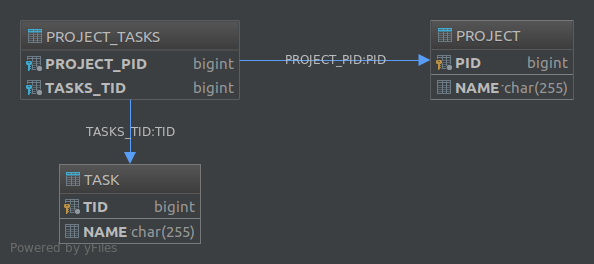
public void setName(String name) {

this.name = name;

}

}

This will create the following database structure:

[](https://i.stack.imgur.com/TDcoI.png)

### The @JoinTable annotation also lets you customize various aspects of the join table. For example, had we annotated the tasks property like this:

@JoinTable(

name = "MY\_JT",

joinColumns = @JoinColumn(

name = "PROJ\_ID",

referencedColumnName = "PID"

),

inverseJoinColumns = @JoinColumn(

name = "TASK\_ID",

referencedColumnName = "TID"

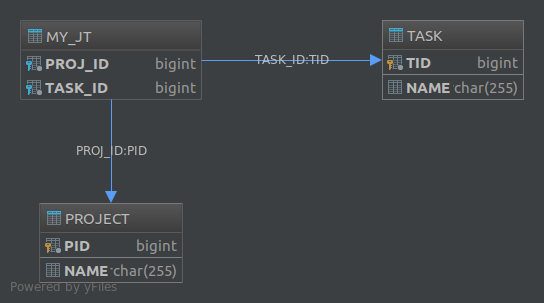
)

)

@OneToMany

private List<Task> tasks;

The resulting database would have become:

[](https://i.stack.imgur.com/WhC82.png)

Finally, if you want to create a schema for a many-to-many association, using a join table is the only available solution.

## @Basic

### A basic attribute is one where the attribute class is a simple type such as String, Number, Date or a primitive. A basic attribute's value can map directly to the column value in the database. The following table summarizes the basic types and the database types they map to.

|  |  |
| --- | --- |
| Java type | Database type |
| String (char, char[]) | VARCHAR (CHAR, VARCHAR2, CLOB, TEXT) |
| Number (BigDecimal, BigInteger, Integer, Double, Long, Float, Short, Byte) | NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE) |
| int, long, float, double, short, byte | NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE) |
| byte[] | VARBINARY (BINARY, BLOB) |
| boolean (Boolean) | BOOLEAN (BIT, SMALLINT, INT, NUMBER) |
| java.util.Date | TIMESTAMP (DATE, DATETIME) |
| java.sql.Date | DATE (TIMESTAMP, DATETIME) |
| java.sql.Time | TIME (TIMESTAMP, DATETIME) |
| java.sql.Timestamp | TIMESTAMP (DATETIME, DATE) |
| java.util.Calendar | TIMESTAMP (DATETIME, DATE) |
| java.lang.Enum | NUMERIC (VARCHAR, CHAR) |
| java.util.Serializable | VARBINARY (BINARY, BLOB) |

## @Column and difference between @Basic and @ Column??

### This represents column in the database table.

### @Basic signifies that an attribute is to be persisted and a standard mapping is to be used. It has parameters which allow you to specify whether the attribute is to be lazily loaded and whether it's nullable.

### @Column allows you to specify the name of the column in the database to which the attribute is to be persisted.

# JPA Sample Example

Plain JPA - <https://www.javaguides.net/2018/11/jpa-with-hibernate-5-bootstrapping-example.html>

<https://www.tutorialspoint.com/jpa/jpa_entity_managers.htm>

# CORE of JPA

## Entities:

### An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table.

### An entity class must follow these requirements.

### The class must be annotated with the javax.persistence.Entity annotation.

### The class must have a public or protected, no-argument constructor. The class may have other constructors.

### The class must not be declared final.

### No methods or persistent instance variables must be declared final.

### If an entity instance is passed by value as a detached object, such as through a session bean’s remote business interface, the class must implement the Serializable interface.

### Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.

### Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class’s methods. Clients must access the entity’s state through accessor or business methods.

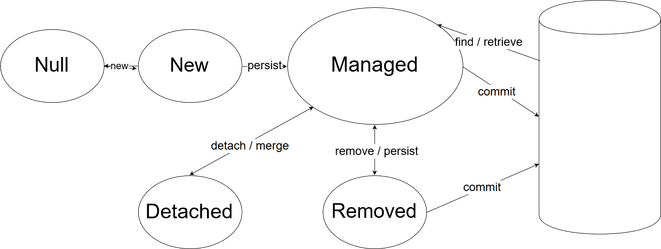
## Entity LifeCycle

### In persistence life cycle, the object lies in the following states: -

### **Transient** - The object is called to be in the transient state when it is just declared by using the new keyword. When an object remains in the transient state, it doesn't contain any identifier(primary key) in the database.

### **Persistence** - In this state, an object is associated with the session and either saved to a database or retrieved from the database. When an object remains in the persistence state, It contains a row of the database and consists of an identifier value. We can make an object persistent by associating it with the hibernate session.

### **Detached** - The object enters into a detached state when the hibernate session is closed. The changes made to the detached objects are not saved to the database.



### As a reminder, the purpose of the EntityManager is to the relationship between the JPA entity and the underlying datasource.

### The above diagram shows the 5 key stages of JPA entity management:

### Object Doesn't Exist – This is a null object

### MyObject myObject = null;

### New Object – Not associated with the EntityManager, and doesn't exist on database

### MyObject myObject = new MyObject();

### Managed – This is the stage where the object becomes persisted and managed by the EntityManager. To do this, we need to call the persist method from within a transaction. The object is then persisted to the database when the commit method is called

### entityManager.getTransaction().begin();

### MyObject myObject = new MyObject();

### entityManager.persist(myObject);

### entityManager.getTransaction().commit();

### Detached – This state removes the object from the EntityManager, but the object **still** exists on the database. Some EntityManager methods on a detached object will result in an IllegalArgumentException. The object can be reattached to the EntityManager through the merge method

### entityManager.detach(myObject);

### Removed – Deletes the object from the database. Like persist, this also needs to take place inside a transaction.

### entityManager.getTransaction().begin();

### entityManager.removed(myObject);

### entityManager.getTransaction().commit();

## Entity callback methods and event listener

### The EntityListeners annotation allows you to set a Listener class that will contain event handling methods (callback methods) defined by the Entity or Mapped Superclass.

### Callback methods are used to call on certain Entity events (that is, add processing, for example, deletion of Entity methods by JPA), can be added to the entity class, to the mapped superclass, or to the callback listener class specified by the EntityListeners annotation (see previous question). There are seven callback methods (and annotations with the same names):

### 1) PrePersist

### 2) PostPersist

### 3) PreRemove

### 4) PostRemove

### 5) PreUpdate

### 6) PostUpdate

### 7) PostLoad

## How to persist entity to databases?

### The following steps are performed to persist an entity object.

### Create Entity class

### Create persistence.xml / Persistence configuration

### Write code to persist entity.

ENTITY CLASS

1. @Entity
2. @Table(name="student")
3. **public** **class** Student {
5. @Id
6. **private** String s\_name;

Persistence.xml (not needed in spring boot)

1. <persistence>
2. <persistence-unit name="Student\_details">
4. <**class**>com.javatpoint.jpa.student.StudentEntity</**class**>
6. <properties>
7. // GIVE PROVIDER NAME
8. <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
9. <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/studentdata"/>
10. <property name="javax.persistence.jdbc.user" value="root"/>
11. <property name="javax.persistence.jdbc.password" value=""/>
12. <property name="eclipselink.logging.level" value="SEVERE"/>
13. <property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
14. </properties>
16. </persistence-unit>
17. </persistence>

CODE TO PERSIST

### Create an entity manager factory object. The **EntityManagerFactory** interface present in **java.persistence** package is used to provide an entity manager.

### EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student\_details");

### Obtain an entity manager from the factory.

### EntityManager em=emf.createEntityManager();

### Initialize an entity manager.

### em.getTransaction().begin();

### Persist the data into the relational database.

### em.persist(s1);

### Closing the transaction

### em.getTransaction().commit();

### Release the factory resources.

### emf.close();

### 

## Entity manager methods to find, delete, insert and update?

### An entity manager is responsible for the following operations.

### The entity manager implements the API and encapsulates all of them within a single interface.

### The entity manager is used to read, delete and write an entity.

### An object referenced by an entity is managed by entity manager.

### **To delete an entity , you must first called find.**

## Access Strategy in JPA

### The access strategy defines how your JPA implementation, e.g., Hibernate or EclipseLink, access your entity attributes. You can choose between field-based access and property-based access:

### If you use field-based access, your JPA implementation uses reflection to read or write your entity attributes directly. It also expects that you place your mapping annotations on your entity attributes.

### If you use property-based access, you need to annotate the getter methods of your entity attributes with the required mapping annotations. Your JPA implementation then calls the getter and setter methods to access your entity attributes.

### By default, you specify the access strategy implicitly by annotating your [primary key attribute](https://thorben-janssen.com/jpa-generate-primary-keys/) or its getter method with an @Id annotation. If you annotate the attribute itself, Hibernate uses field-based access.

@Id

protected Long id;

PROPERTY BASED:

@Id

     public Long getId() {

        return id;

    }

## How to override default strategy and can we mix both strategy in one entity?

### Yes, we can use both type in one entity.

To override use @Access annotation –

@Version

    @Access(AccessType.FIELD)

    private int version;

    @Id

    public Long getId() {

        return id;

    }

## Which access strategy we should use?

### Field based strategy is always better, here are 5 advantages of field based-

<https://thorben-janssen.com/access-strategies-in-jpa-and-hibernate/>

1. Readable code
2. No need of getter setter
3. No need to mark utility method as transient
4. Avoid bugs when working with proxies
5. Flexible and custom implementation of getter and setter is possible using FIELD.

## How to define Primary key and how to create composite Primary key?

### Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number. The unique identifier, or **primary key**, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

### Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field.

### Composite primary keys are used when a primary key consists of more than one attribute, which corresponds to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the javax.persistence.EmbeddedId and javax.persistence.IdClass annotations.

### Floating-point types should never be used in primary keys. If you use a generated primary key, only integral types will be portable.

## How to define composite Primary key?

<https://www.baeldung.com/jpa-composite-primary-keys>

### A composite primary key – also called a composite key – is a combination of two or more columns to form a primary key for a table.

### In JPA, we have two options to define the composite keys: The *@IdClass* and *@EmbeddedId* annotations.

### **In order to define the composite primary keys, we should follow some rules:**

### The composite primary key class must be public

### It must have a no-arg constructor

### It must define *equals()* and *hashCode()* methods

### It must be *Serializable*

Both ways are described in below 2 questions.

## @IDClass annotation

### Let's say we have a table called Account and it has two columns – accountNumber, accountType – that form the composite key. Now we have to map it in JPA. As per the JPA specification, let's create an AccountId class with these primary key fields:

|  |  |
| --- | --- |
|  | **public** **class** AccountId **implements** Serializable { |
|  | **private** String accountNumber; |
|  |  |
|  | **private** String accountType; |
|  |  |
|  | // default constructor |
|  |  |
|  | **public** AccountId(String accountNumber, String accountType) { |
|  | **this**.accountNumber = accountNumber; |
|  | **this**.accountType = accountType; |
|  | } |
|  |  |
|  | // equals() and hashCode() |
|  | } |

### Next, let's associate the AccountId class with the entity Account.

### In order to do that, we need to annotate the entity with the [@IdClass](https://www.baeldung.com/hibernate-identifiers) annotation. We must also declare the fields from the AccountId class in the entity Account and annotate them with @Id:

|  |  |
| --- | --- |
|  | @Entity |
|  | @IdClass(AccountId.class) |
|  | **public** **class** Account { |
|  | @Id |
|  | **private** String accountNumber; |
|  |  |
|  | @Id |
|  | **private** String accountType; |
|  |  |
|  | // other fields, getters and setters |
|  | } |

## @EmbeddedId

### Let's consider another example where we have to persist some information of a Book with title and language as the primary key fields.

### In this case, the primary key class, **BookId, must be annotated with**@Embeddable:

|  |  |
| --- | --- |
|  | @Embeddable |
|  | **public** **class** BookId **implements** Serializable { |
|  | **private** String title; |
|  | **private** String language; |
|  |  |
|  | // default constructor |
|  |  |
|  | **public** BookId(String title, String language) { |
|  | **this**.title = title; |
|  | **this**.language = language; |
|  | } |
|  |  |
|  | // getters, equals() and hashCode() methods |
|  | } |

### Then, we need to embed this class in the Book entity using [@EmbeddedId](https://www.baeldung.com/jpa-many-to-many):

|  |  |
| --- | --- |
|  | @Entity |
|  | **public** **class** Book { |
|  | @EmbeddedId |
|  | **private** BookId bookId; |
|  |  |
|  | // constructors, other fields, getters and setters |
|  | } |

## @IdClass vs @EmbeddedId

### As we just saw, the difference on the surface between these two is that with *@IdClass*, we had to specify the columns twice – once in *AccountId* and again in *Account.*But, with *@EmbeddedId*we didn't.

### There are some other tradeoffs, though.

### **For example, these different structures affect the JPQL queries that we write.**

### For example, with *@IdClass*, the query is a bit simpler:

|  |  |
| --- | --- |
|  | SELECT account.accountNumber FROM Account account |

### With *@EmbeddedId*, we have to do one extra traversal:

|  |  |
| --- | --- |
|  | SELECT book.bookId.title FROM Book book |

### Also, ***@IdClass* can be quite useful in places where we** **are using a composite key class that we can't modify.**

### Finally, if we're going to access parts of the composite key individually, we can make use of *@IdClass,*but **in places where we frequently use the complete identifier as an object, *@EmbeddedId* is preferred.**

## What are the requirements for Primary Key class?

### A primary key class must meet these requirements.

### The access control modifier of the class must be public.

### The properties of the primary key class must be public or protected if property-based access is used.

### The class must have a public default constructor.

### The class must implement the hashCode() and equals(Object other) methods.

### The class must be serializable.

### A composite primary key must be represented and mapped to multiple fields or properties of the entity class or must be represented and mapped as an embeddable class.

### If the class is mapped to multiple fields or properties of the entity class, the names and types of the primary key fields or properties in the primary key class must match those of the entity class.

## What is the Embeddable class?

### An embeddable class is a class that is not used by itself, only as part of one or more Entity classes. An entity class can contain both single embedded classes and collections of such classes. Also such classes can be used as keys or map values. At run time, each embedded class belongs to only one object of the Entity class and cannot be used to transfer data between the objects of the Entity classes (that is, such a class is not a common data structure for different objects). In general, such a class serves to make the definition of common attributes for several Entity, we can assume that JPA simply embeds into the Entity instead of an object of this class, the attributes it contains.

### Embeddable classes represent the state of an entity but do not have a persistent identity of their own. The objects of such classes share the identity of the entity classes that owns it. An Entity may have single-valued or multivalued embeddable class attributes.

## Can the Embeddable class contain another Embeddable class?

### Yes maybe.

## Can the Embeddable class contain relationships with other Entity or Entity collections? If it can, are there any restrictions on such relationships?

### Maybe, but only if such a class is not used as the primary key or the map key.

## What requirements does JPA set for Embeddable classes?

### Such classes must satisfy the same rules as the Entity classes, except that they do not have to contain a primary key and be marked with the Entity annotation (see question 10),

### The Embeddable class must be marked with the Embeddable annotation or described in the XML configuration file JPA.

## Direction in entities

### The direction of a relationship can be either bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

### **Bidirectional Relationships**

### In a **bidirectional** relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class’s code can access its related object. If an entity has a related field, the entity is said to “know” about its related object. For example, if Order knows what LineItem instances it has and if LineItem knows what Order it belongs to, they have a bidirectional relationship.

### Bidirectional relationships must follow these rules.

### The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.

### The many side of many-to-one bidirectional relationships must not define the mappedBy element. **The many side is always the owning side of the relationship.**

### For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.

### For many-to-many bidirectional relationships, either side may be the owning side.

### **Unidirectional Relationships**

### In a **unidirectional** relationship, only one entity has a relationship field or property that refers to the other. For example, LineItem would have a relationship field that identifies Product, but Product would not have a relationship field or property for LineItem. In other words, LineItem knows about Product, but Product doesn’t know which LineItem instances refer to it.

### **Queries and Relationship Direction**

### Java Persistence query language and Criteria API queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one entity to another. For example, a query can navigate from LineItem to Product but cannot navigate in the opposite direction. For Order and LineItem, a query could navigate in both directions because these two entities have a bidirectional relationship.

## Cascade Operations and Relationships

### Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade delete relationship.

### The javax.persistence.CascadeType enumerated type defines the cascade operations that are applied in the cascade element of the relationship annotations. [Table 32-1](https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html#gjjnj) lists the cascade operations for entities.

**Table 32-1 Cascade Operations for Entities**

|  |  |
| --- | --- |
| **Cascade Operation** | **Description** |
| ALL | All cascade operations will be applied to the parent entity’s related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE} |
| DETACH | If the parent entity is detached from the persistence context, the related entity will also be detached. |
| MERGE | If the parent entity is merged into the persistence context, the related entity will also be merged. |
| PERSIST | If the parent entity is persisted into the persistence context, the related entity will also be persisted. |
| REFRESH | If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed. |
| REMOVE | If the parent entity is removed from the current persistence context, the related entity will also be removed. |

### Cascade delete relationships are specified using the cascade=REMOVE element specification for @OneToOne and @OneToMany relationships. For example:

### @OneToMany(cascade=REMOVE, mappedBy="customer")

### public Set<Order> getOrders() { return orders; }

## Multiplicity Entity Relationships – One to one

When an object is mapped to only one other object. As Student->Passport

### Example: We have two entities here: Employee and Account.

### **One Employee can have only one Account**. Similarly, an account will be associated with one employee only. It’s one to one relationship for this example.

<https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping/>

One to one can be unidirectional or bidirectional .

**In unidirectional association,**the source entity has a relationship field that refers to the target entity and the source entity’s table contains the foreign key.  
**In a bidirectional association,**each entity (i.e. source and target) has a relationship field that refers to each other and the target entity’s table contains the foreign key. The source entity must use the mappedBy attribute to define the bidirectional one-to-one mapping.

We will discuss 4 different variations.

Table of Contents

[Various supported techniques for one to one mapping](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping/#techniques)

1. [Using foreign key association](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping/#1)

2. [Using join table](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping/#2)

3. [Using shared primary key](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping/#3)

4. [@MapsId](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping/#4)

Either way we have to use [**@OneToOne**](https://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html) annotation.

1. First technique is most widely used and uses a **foreign key column in one of the tables**.
2. Second technique uses a rather known solution of having a **third table to store mapping** between first two tables.
3. Third technique is something new which uses a **common primary key value in both the tables**. - In this technique, hibernate will ensure that it will use a common primary key value in both the tables. This way primary key of EmployeeEntity can safely be assumed the primary key of AccountEntity also.

We need to define the owning relationship. The owner side of the relationship.

In database, owner table contains extra column, which is foreign key. And is Java side Owner contains @JoinColumn and other annotations. Absence of mappedBy is defined as owning side.

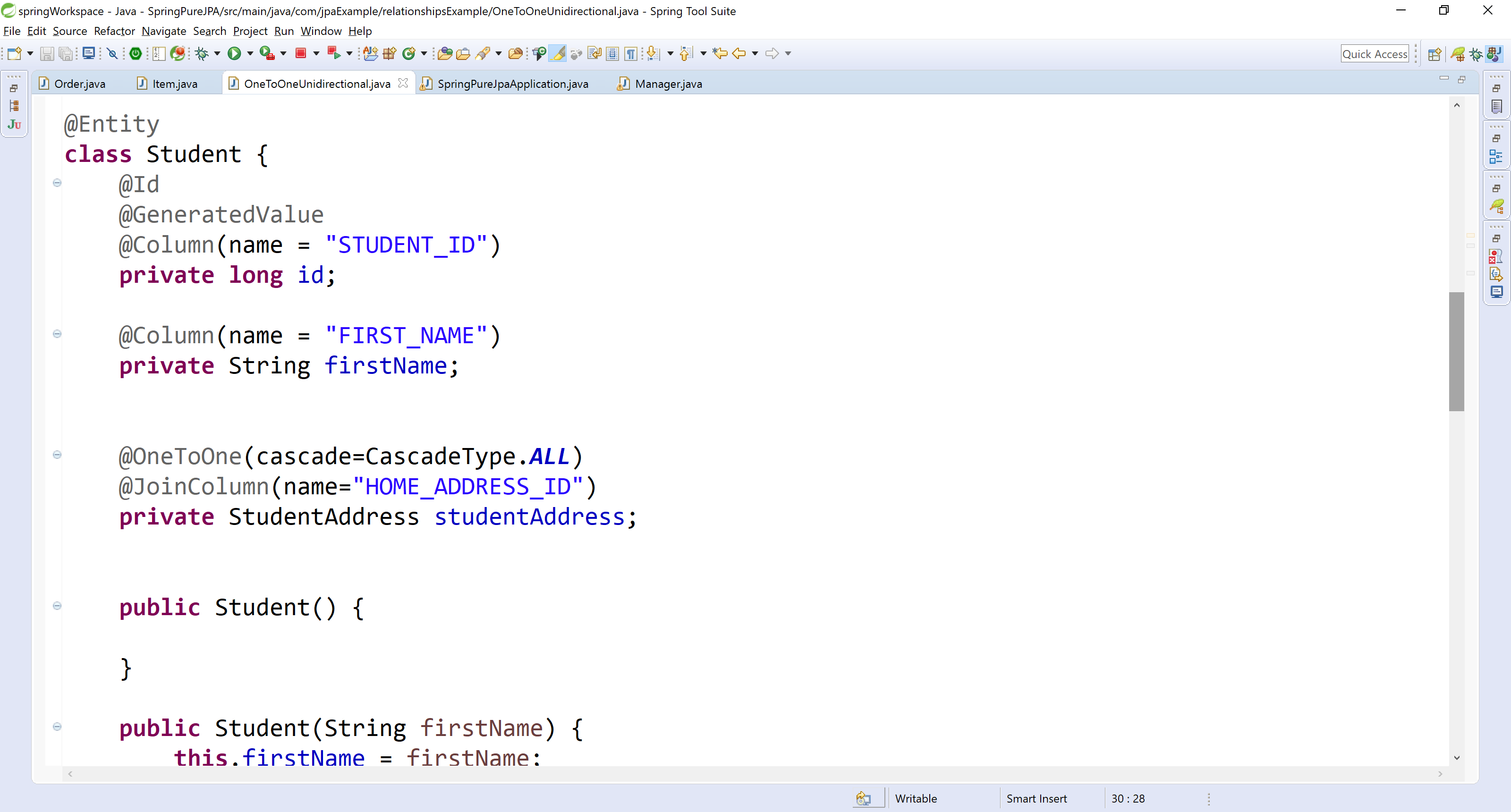
We need to have @OneToOne on non-owning side to ensure Bidirectional relationship.

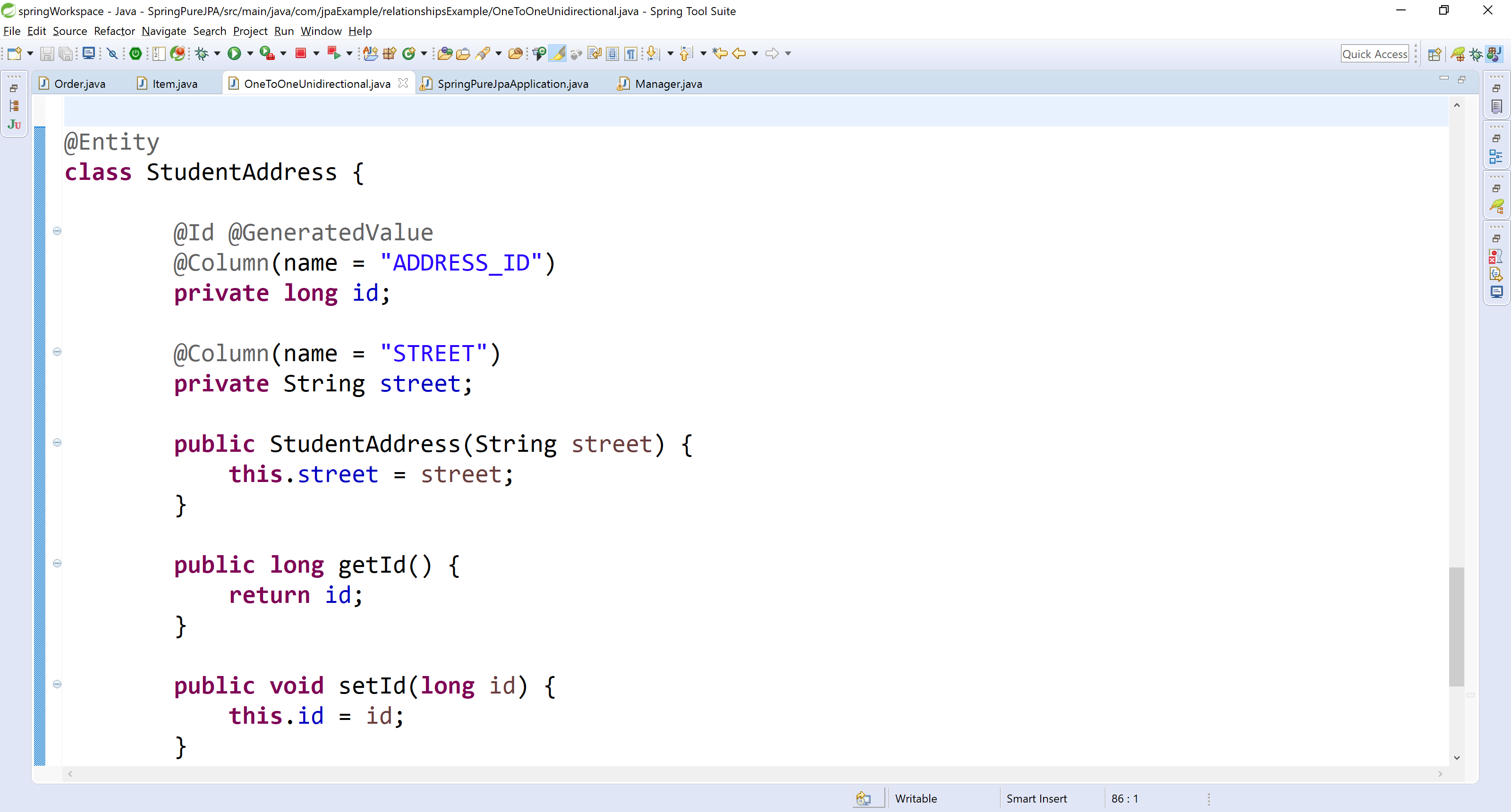
|  |  |  |
| --- | --- | --- |
| Type | employeeEntity | AccountEntity |
| Foreign key association | @OneToOne  @JoinColumn(name="ACCOUNT\_ID")  private AccountEntity account; | @OneToOne(mappedBy="account")  private EmployeeEntity employee; |
| Join table | @OneToOne(cascade = CascadeType.ALL)  @JoinTable(name="EMPLOYEE\_ACCCOUNT", joinColumns = @JoinColumn(name="EMPLOYEE\_ID"),  inverseJoinColumns = @JoinColumn(name="ACCOUNT\_ID"))  private AccountEntity account; | @OneToOne(mappedBy="account")  private EmployeeEntity employee; |
| Common Primary key | @OneToOne(cascade = CascadeType.ALL)  @PrimaryKeyJoinColumn  private AccountEntity account; | @OneToOne(mappedBy="account", cascade=CascadeType.ALL)  private EmployeeEntity employee; |
| MapsId | @OneToOne  @MapsId  private AccountEntity account; | // no GeneratedValue issued on @Id - This way, the id property serves as both Primary Key and Foreign Key. You’ll notice that the @Id column no longer uses a @GeneratedValue annotation since the identifier is populated with the identifier of the employee association. |

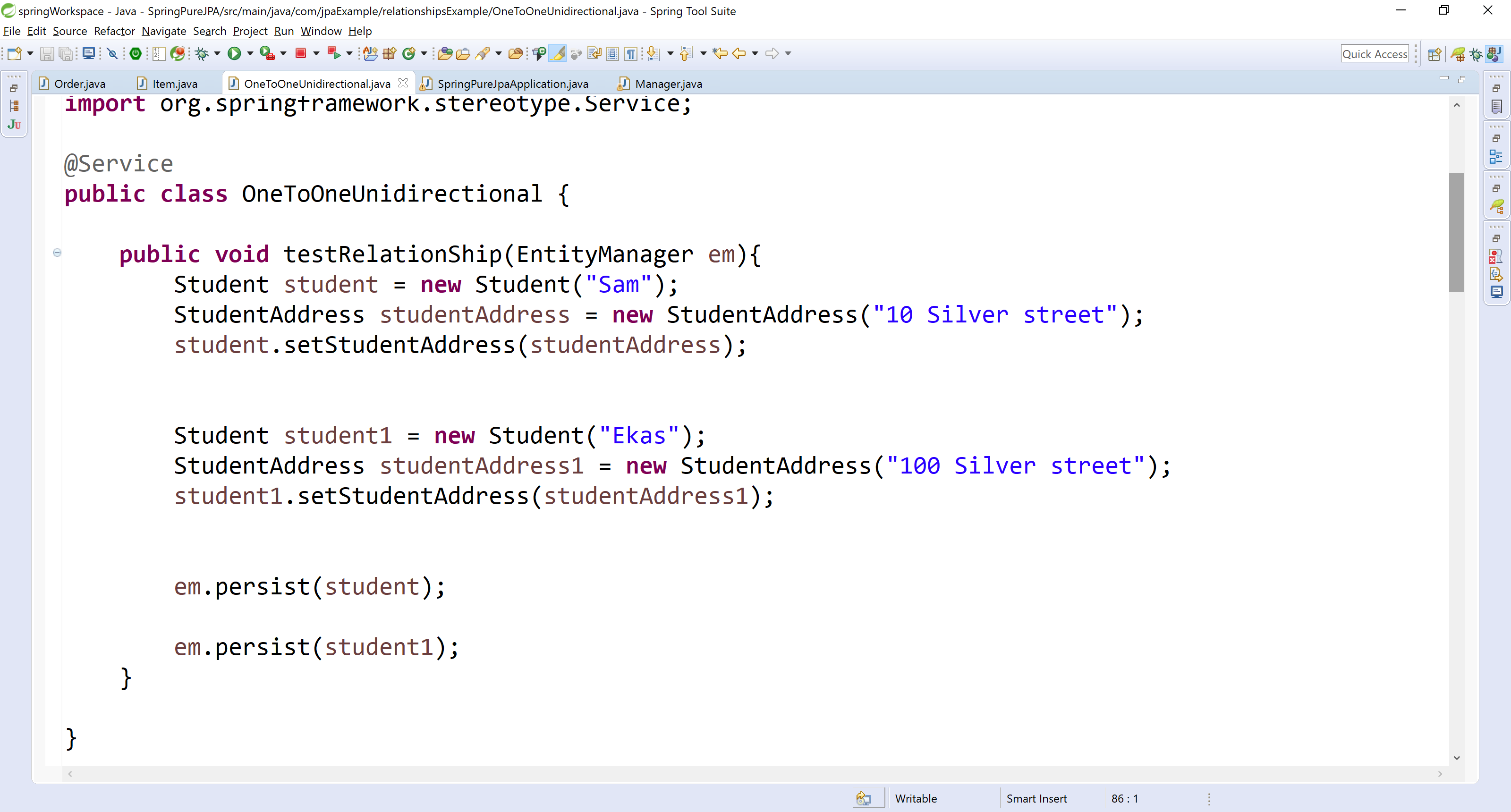
<https://vladmihalcea.com/the-best-way-to-map-a-onetoone-relationship-with-jpa-and-hibernate/>

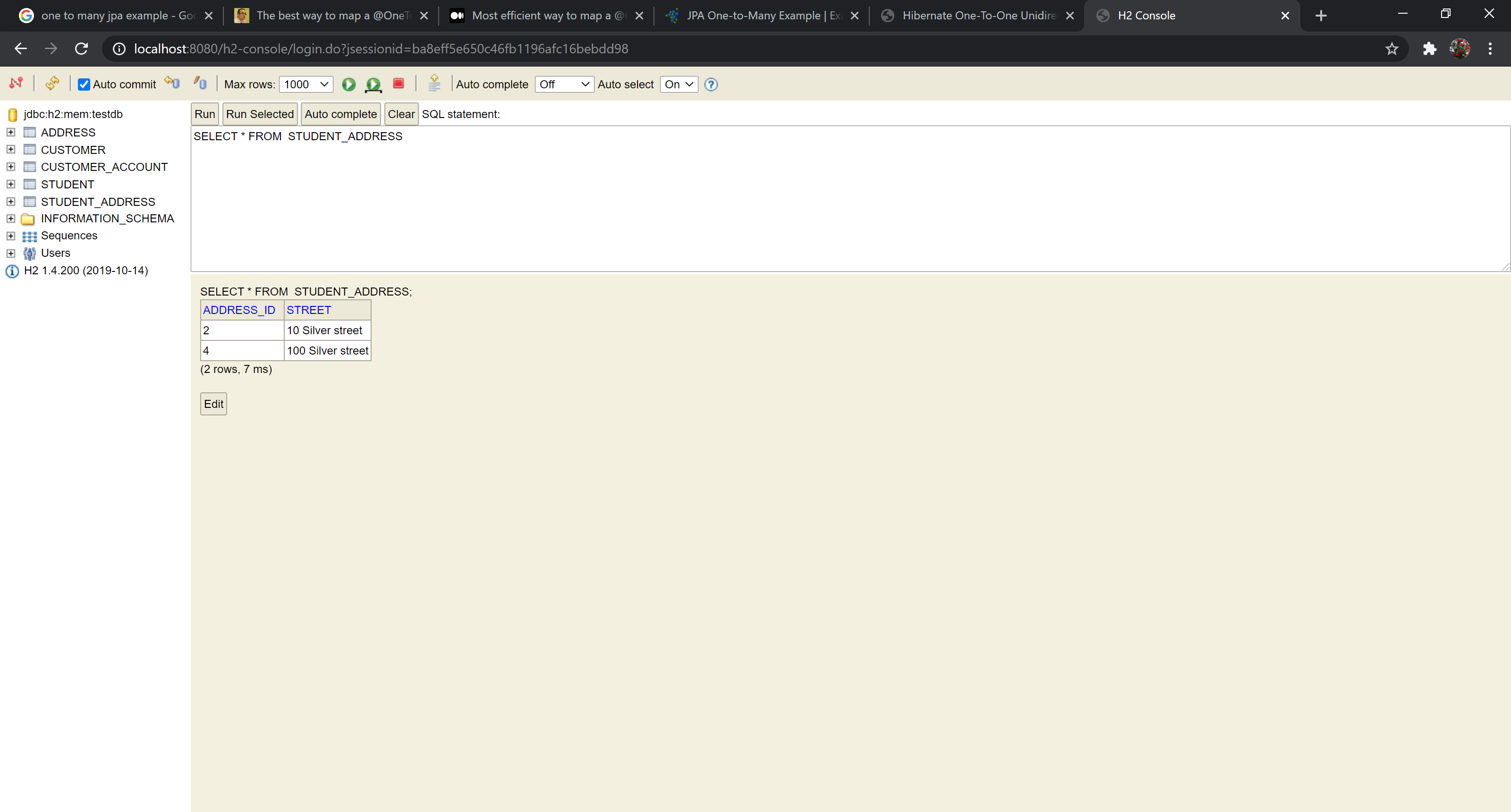
<https://www.callicoder.com/hibernate-spring-boot-jpa-one-to-one-mapping-example/>

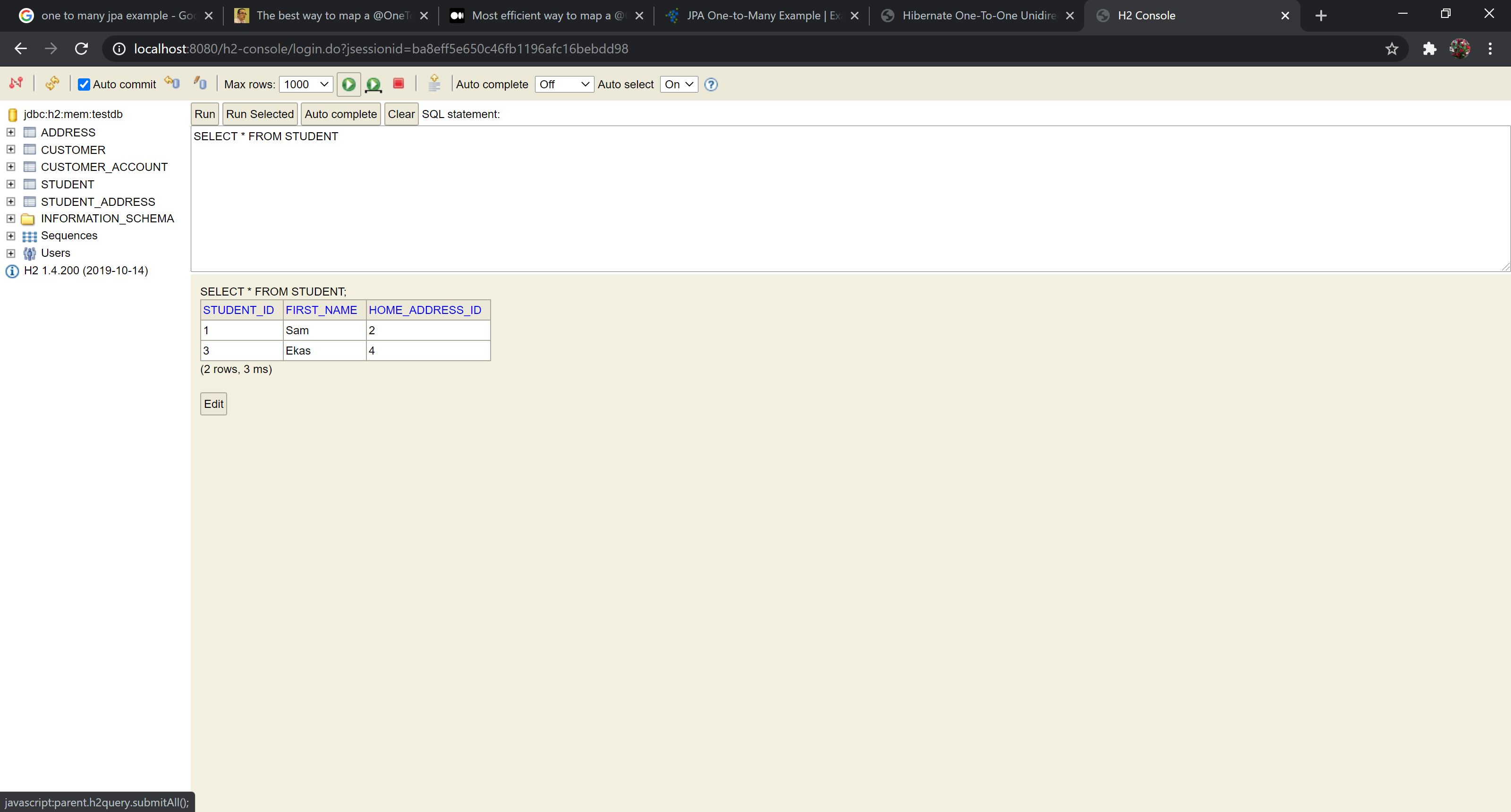
## OnetoOne unidirectional example.











<http://websystique.com/hibernate/hibernate-one-to-one-unidirectional-with-foreign-key-associations-annotation-example/>

## One to Many / Many to one

### Use one to mapping to create **1..N relationship** between entities or objects.

### For example, we have to write two entities i.e. EmployeeEntity and AccountEntity such that multiple accounts can be associated with a single employee, but one single account can not be shared between two or more employees.

### Example – a course can have multiple reviews. But review can be associated with Single course.

### This problem can be solved in two different ways.

### One is to have a **foreign key column** in account table i.e. EMPLOYEE\_ID. This column will refer to primary key of Employee table. This way no two accounts can be associated with multiple employees. Obviously, account number needs to be unique for enforcing this restriction.

### Second approach is to have a common **join table** lets say EMPLOYEE\_ACCOUNT. This table will have two column i.e. EMP\_ID which will be foreign key referring to primary key in EMPLOYEE table and similarly ACCOUNT\_ID which will be foreign key referring to primary key of ACCOUNT table

## Many to many

A student can have multiple course and course can have multiple student. This is only possible by @JoinTable. Direction does not matter in this case.

## Orphan removal

### When a target entity in one-to-one or one-to-many relationship is removed from the relationship, it is often desirable to cascade the remove operation to the target entity. Such target entities are considered “orphans,” and the orphanRemoval attribute can be used to specify that orphaned entities should be removed. For example, if an order has many line items and one of them is removed from the order, the removed line item is considered an orphan. If orphanRemoval is set to true, the line item entity will be deleted when the line item is removed from the order.

### The orphanRemoval attribute in @OneToMany and @oneToOne takes a Boolean value and is by default false.

### The following example will cascade the remove operation to the orphaned order entity when the customer entity is deleted:

### @OneToMany(mappedBy="customer", orphanRemoval="true")

### public List<Order> getOrders() { ... }

### JPA 2 supports an additional and more aggressive remove cascading mode which can be specified using the orphanRemoval element of the [@OneToOne](https://www.objectdb.com/api/java/jpa/OneToOne) and [@OneToMany](https://www.objectdb.com/api/java/jpa/OneToMany) annotations:

### [@Entity](https://www.objectdb.com/api/java/jpa/Entity)

### class Employee {

### :

### [@OneToOne](https://www.objectdb.com/api/java/jpa/OneToOne)([orphanRemoval](https://www.objectdb.com/api/java/jpa/OneToOne/orphanRemoval)=true)

### private Address address;

### :

### }

## Difference between orphanRemoval and cascade.remove?

### When an Employee entity object is removed the remove operation is cascaded to the referenced Address entity object. In this regard, orphanRemoval=true and cascade=CascadeType.REMOVE are identical, and if orphanRemoval=true is specified, CascadeType.REMOVE is redundant.

### The difference between the two settings is in the response to disconnecting a relationship. For example, such as when setting the address field to null or to another Address object.

### If orphanRemoval=true is specified the disconnected Address instance is automatically removed. This is useful for cleaning up dependent objects (e.g. Address) that should not exist without a reference from an owner object (e.g. Employee).

### If only cascade=CascadeType.REMOVE is specified no automatic action is taken since disconnecting a relationship is not a remove operation.

### To avoid dangling references as a result of orphan removal this feature should only be enabled for fields that hold private non shared dependent objects.

### Orphan removal can also be set for collection and map fields. For example:

### [@Entity](https://www.objectdb.com/api/java/jpa/Entity)

### class Employee {

### :

### [@OneToMany](https://www.objectdb.com/api/java/jpa/OneToMany)([orphanRemoval](https://www.objectdb.com/api/java/jpa/OneToMany/orphanRemoval)=true)

### private List<Address> addresses;

### :

### }

### In this case, removal of an Address object from the collection leads to automatic removal of that object from the database.

## Inheritance in entity

### Inheritance is one of the key concepts in Java, and it’s used in most domain models. That often becomes an issue, if you try to map these models to a relational database. SQL doesn’t support this kind of relationship and Hibernate, or any other JPA implementation has to map it to a supported concept.

### You can choose between 4 strategies that map the inheritance structure of your domain model to different table structures. Each of these strategies has its advantages and disadvantages. It’s, therefore, important to understand the different concepts and to choose the one that fits best.

### [Mapped Superclass](https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/#Mapped_Superclass)

### [Table per Class](https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/#Table_per_Class)

### [Single Table](https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/#Single_Table)

### [Joined](https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/#Joined)

### Domain Model

### I will use the same simple domain model in all of the examples to show you the different inheritance strategies. It consists of an author who has written different kinds of publications. A publication can either be a book or a blog post. Both of them share most of their attributes, like the id, a title, and a publishing date. In addition to the shared attributes, the book also stores the number of pages, and the blog post persists its URL.

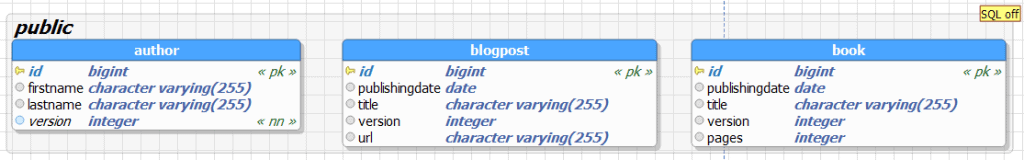
### 

### 4 Inheritance Strategies

### JPA and Hibernate support 4 inheritance strategies which map the domain objects to different table structures.

### **Mapped Superclass**

### The mapped superclass strategy is the simplest approach to mapping an inheritance structure to database tables. It maps each concrete class to its own table.

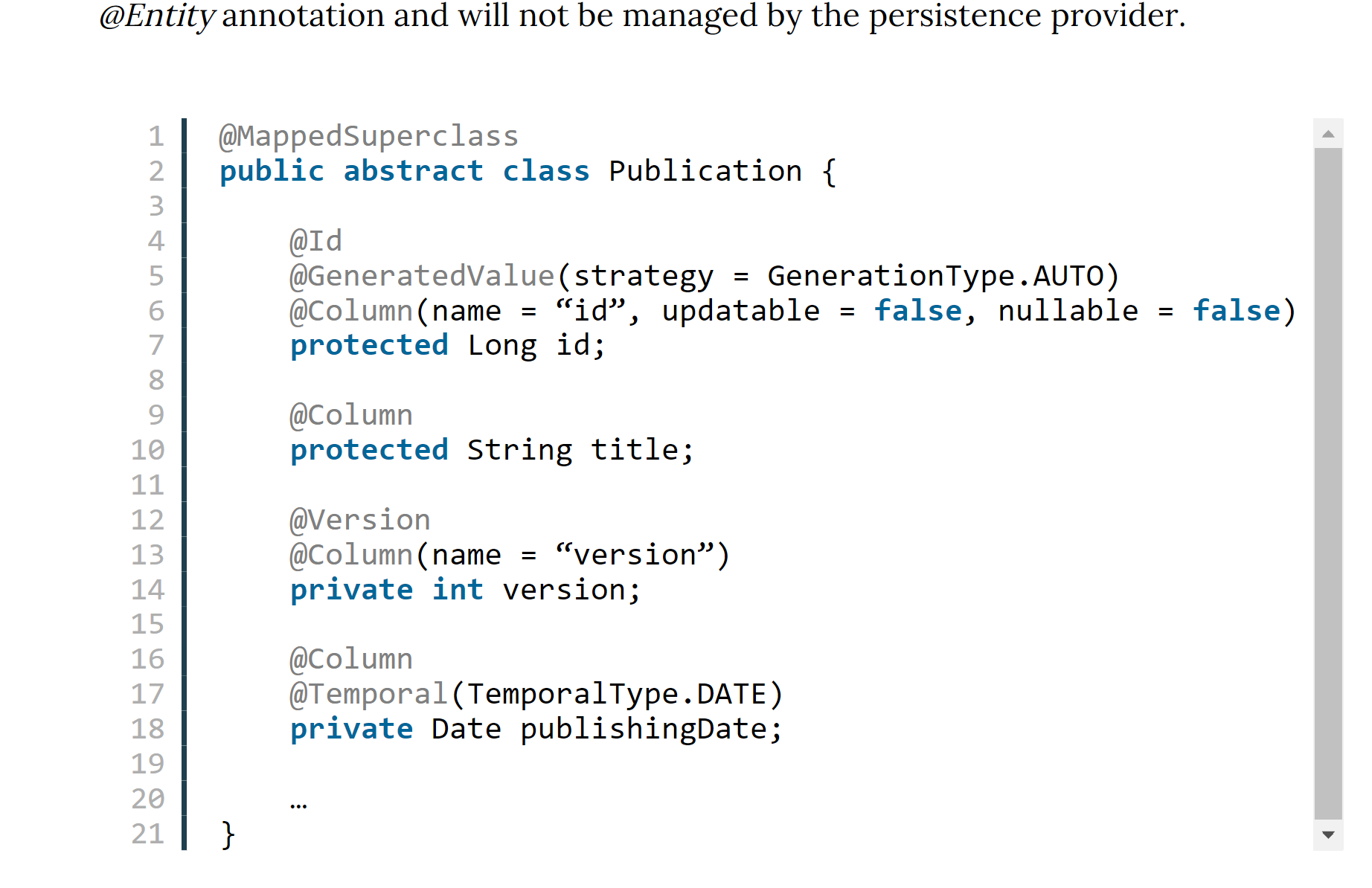


### That allows you to share the attribute definition between multiple entities. But it also has a huge drawback. A mapped superclass is not an entity, and there is no table for it.

### **Drawback**: That means that you can’t use polymorphic queries that select all Publication entities and you also can’t define a relationship between an Author entity and all Publications. You either need to use uni-directional relationship from the Publication to the Author entity, or you have to define a relationship between an Author and each kind of Publication. In general, if you need these relationships, you should have a look at the other inheritance strategies. They are most likely a better fit for your use case.

### **When to use:** If you just want to share state and mapping information between your entities, the mapped superclass strategy is a good fit and easy to implement. You just have to set up your inheritance structure, annotate the mapping information for all attributes and add the @MappedSuperclass annotation to your superclass. Without the @MappedSuperclass annotation, Hibernate will ignore the mapping information of your superclass.

### **EXAMPLE** : You can see an example of such a mapping in the following code snippets. the Publication class is annotated with @MappedSuperclass and provides the shared attributes with their mapping annotations. As you can see, Publication has no @Entity annotation and will not be managed by the persistence provider.



### The subclasses Book and BlogPost extend the Publication class and add their specific attributes with their mapping annotations. Both classes are also annotated with @Entity and will be managed by the persistence provider.

### 

### As I explained at the beginning of this section, you can’t use the inheritance structure for polymorphic queries or to define relationships. But you can, of course, query the entities in the same way as any other entity.

|  |  |
| --- | --- |
| 1 | List books = em.createQuery(“SELECT b FROM Book b”, Book.class).getResultList(); |

### The Book entity and all its attributes are mapped to the book table. This makes the generated query simple and efficient. It just has to select all columns of the book table.

|  |  |
| --- | --- |
| 1 | 15:38:36,020 DEBUG [org.hibernate.SQL] – select book0\_.idas id1\_2\_, book0\_.publishingDate as publishi2\_2\_,book0\_.title as title3\_2\_,book0\_.version as version4\_2\_,book0\_.pages as pages5\_2\_ from Book book0\_ |

### **Table per Class**

### The table per class strategy is similar to the mapped superclass strategy. The main difference is that the superclass is now also an entity. Each of the concrete classes gets still mapped to its own database table. This mapping allows you to use polymorphic queries and to define relationships to the superclass. But the table structure adds a lot of complexity to polymorphic queries, and you should, therefore, avoid them.

### 

### The definition of the superclass with the table per class strategy looks similar to any other entity definition. You annotate the class with @Entity and add your mapping annotations to the attributes. The only difference is the additional @Inheritance annotation which you have to add to the class to define the inheritance strategy. In this case, it’s the InheritanceType.TABLE\_PER\_CLASS.



### The definitions of the Book and BlogPost entities are identical to the previously discussed mapped superclass strategy. You just have to extend the Publication class, add the @Entity annotation and add the class specific attributes with their mapping annotations.

|  |  |
| --- | --- |
| 12345678 | @Entity(name = “Book”)public class Book extends Publication {    @Column    private int pages;    …} |
| 12345678 | @Entity(name = “BlogPost”)public class BlogPost extends Publication {    @Column    private String url;    …} |

### The table per class strategy maps each entity to its own table which contains a column for each entity attribute. That makes the query for a specific entity class easy and efficient.

|  |  |
| --- | --- |
| 1 | List books = em.createQuery(“SELECT b FROM Book b”, Book.class).getResultList(); |
| 1 | 15:56:21,463 DEBUG [org.hibernate.SQL] – select book0\_.id as id1\_3\_, book0\_.publishingDate as publishi2\_3\_, book0\_.title as title3\_3\_, book0\_.version as version4\_3\_, book0\_.pages as pages1\_2\_ from Book book0\_ | |

### The superclass is now also an entity and you can, therefore, use it to define a relationship between the Author and the Publication entity. This allows you to call the getPublications() method to get all Publications written by that Author. Hibernate will map each Publication to its specific subclass.

|  |  |
| --- | --- |
| 123456789 | List authors= em.createQuery(“SELECT a FROM Author a”, Author.class).getResultList();for (Author a : authors) {    for (Publication p : a.getPublications()) {        if (p instanceof Book)        log(p.getTitle(), “book”);        else        log(p.getTitle(), “blog post”);    }} |

### The Java code looks easy and comfortable to use. But if you have a look at the generated SQL statement, you recognize that the table model makes the required query quite complicated.

|  |  |
| --- | --- |
| 123 | 15:57:16,722 DEBUG [org.hibernate.SQL] – select author0\_.id as id1\_0\_, author0\_.firstName as firstNam2\_0\_, author0\_.lastName as lastName3\_0\_, author0\_.version as version4\_0\_ from Author author0\_15:57:16,765 DEBUG [org.hibernate.SQL] – select publicatio0\_.authorId as authorId2\_4\_0\_, publicatio0\_.publicationId as publicat1\_4\_0\_, publicatio1\_.id as id1\_3\_1\_, publicatio1\_.publishingDate as publishi2\_3\_1\_, publicatio1\_.title as title3\_3\_1\_, publicatio1\_.version as version4\_3\_1\_, publicatio1\_.pages as pages1\_2\_1\_, publicatio1\_.url as url1\_1\_1\_, publicatio1\_.clazz\_ as clazz\_1\_ from PublicationAuthor publicatio0\_ inner join ( select id, publishingDate, title, version, null::int4 as pages, null::varchar as url, 0 as clazz\_ from Publication union all select id, publishingDate, title, version, pages, null::varchar as url, 1 as clazz\_ from Book union all select id, publishingDate, title, version, null::int4 as pages, url, 2 as clazz\_ from BlogPost ) publicatio1\_ on publicatio0\_.publicationId=publicatio1\_.id where publicatio0\_.authorId=?Effective Java is a book. |

### Hibernate has to join the author table with the result of a subselect which uses a union to get all matching records from the book and blogpost tables. Depending on the amounts of records in both tables, this query might become a performance issue. And it gets even worse if you add more subclasses to the inheritance structure. You should, therefore, try to avoid these kinds of queries or choose a different inheritance strategy.

### **Single Table**

### The single table strategy maps all entities of the inheritance structure to the same database table. This approach makes polymorphic queries very efficient and provides the best performance.

### But it also has some drawbacks. The attributes of all entities are mapped to the same database table. Each record uses only a subset of the available columns and sets the rest of them to null. You can, therefore, not use not null constraints on any column that isn’t mapped to all entities. That can create data integrity issues, and your database administrator might not be too happy about it.

### 

### When you persist all entities in the same table, Hibernate needs a way to determine the entity class each record represents. This is information is stored in a discriminator column which is not an entity attribute. You can either define the column name with a @DiscriminatorColumn annotation on the superclass or Hibernate will use DTYPE as its default name.

### The definition of the subclasses is again similar to the previous examples. But this time, you should also provide a @DiscriminatorValue annotation. It specifies the discriminator value for this specific entity class so that your persistence provider can map each database record to a concrete entity class.

### The @DiscriminatorValue annotation is optional if you use Hibernate. If you don’t provide a discriminator value, Hibernate will use the simple entity name by default. But this default handling isn’t defined by the JPA specification, and you shouldn’t rely on it.

|  |  |
| --- | --- |
| 123456789 | @Entity(name = “Book”)@DiscriminatorValue(“Book”)public class Book extends Publication {    @Column    private int pages;    …} |
| 123456789 | @Entity(name = “BlogPost”)@DiscriminatorValue(“Blog”)public class BlogPost extends Publication {    @Column    private String url;    …} |

### As I explained at the beginning of this section, the single table strategy allows easy and efficient data access. All attributes of each entity are stored in one table, and the query doesn’t require any join statements. The only thing that Hibernate needs to add to the SQL query to fetch a particular entity class is a comparison of the discriminator value. In this example, it’s a simple expression that checks that the column publication\_type contains the value ‘Book‘.

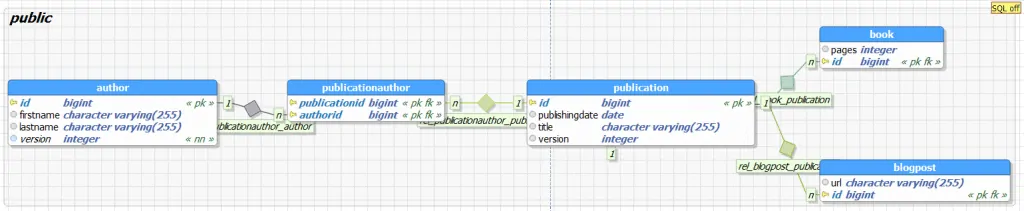
|  |  |
| --- | --- |
| 1 | List books = em.createQuery(“SELECT b FROM Book b”, Book.class).getResultList(); |
| 1 | 16:02:47,411 DEBUG [org.hibernate.SQL] – select book0\_.id as id2\_1\_, book0\_.publishingDate as publishi3\_1\_, book0\_.title as title4\_1\_, book0\_.version as version5\_1\_, book0\_.pages as pages6\_1\_ from Publication book0\_ where book0\_.Publication\_Type=’Book’ | |

### The previously discussed inheritance strategies had their issues with polymorphic queries. They were either not supported or required complex union and join operations. That’s not the case if you use the single table strategy. All entities of the inheritance hierarchy are mapped to the same table and can be selected with a simple query. The following code and log snippets show an example for such a query. As you can see in the log messages, Hibernate selects all columns, including the discriminator column publication\_type, from the publication table. It then uses the discriminator value to select the right entity class and to map the database record. This query is much easier than the one created by the table per class strategy, and you don’t need to worry about performance problems.

|  |  |
| --- | --- |
| 123456789 | List authors= em.createQuery(“SELECT a FROM Author a”, Author.class).getResultList();for (Author a : authors) {    for (Publication p : a.getPublications()) {        if (p instanceof Book)        log(p.getTitle(), “book”);        else        log(p.getTitle(), “blog post”);    }} |
| 123 | 16:04:32,073 DEBUG [org.hibernate.SQL] – select author0\_.id as id1\_0\_, author0\_.firstName as firstNam2\_0\_, author0\_.lastName as lastName3\_0\_, author0\_.version as version4\_0\_ from Author author0\_16:04:32,118 DEBUG [org.hibernate.SQL] – select publicatio0\_.authorId as authorId2\_2\_0\_, publicatio0\_.publicationId as publicat1\_2\_0\_, publicatio1\_.id as id2\_1\_1\_, publicatio1\_.publishingDate as publishi3\_1\_1\_, publicatio1\_.title as title4\_1\_1\_, publicatio1\_.version as version5\_1\_1\_, publicatio1\_.pages as pages6\_1\_1\_, publicatio1\_.url as url7\_1\_1\_, publicatio1\_.Publication\_Type as Publicat1\_1\_1\_ from PublicationAuthor publicatio0\_ inner join Publication publicatio1\_ on publicatio0\_.publicationId=publicatio1\_.id where publicatio0\_.authorId=?Effective Java is a book. | |

### **Joined**

### The joined table approach maps each class of the inheritance hierarchy to its own database table. This sounds similar to the table per class strategy. But this time, also the abstract superclass Publication gets mapped to a database table. This table contains columns for all shared entity attributes. The tables of the subclasses are much smaller than in the table per class strategy. They hold only the columns specific for the mapped entity class and a primary key with the same value as the record in the table of the superclass.



### Each query of a subclass requires a join of the 2 tables to select the columns of all entity attributes. That increases the complexity of each query, but it also allows you to use not null constraints on subclass attributes and to ensure data integrity. The definition of the superclass Publication is similar to the previous examples. The only difference is the value of the inheritance strategy which is InheritanceType.JOINED.



### The definition of the subclasses doesn’t require any additional annotations. They just extend the superclass, provide an @Entity annotation, and define the mapping of their specific attributes.

|  |  |
| --- | --- |
| 12345678 | @Entity(name = “Book”)public class Book extends Publication {    @Column    private int pages;    …} |
| 12345678 | @Entity(name = “BlogPost”)public class BlogPost extends Publication {    @Column    private String url;    …} |

### As I already explained, the columns mapped by each subclass are stored in 2 different database tables. The publication table contains all columns mapped by the superclass Publication and the book table all columns mapped by the Book entity. Hibernate needs to join these 2 tables by their primary keys to select all attributes of the Book entity. This is an overhead that makes these queries slightly slower than the simpler queries generated for the single table strategy.

|  |  |
| --- | --- |
| 1 | List books = em.createQuery(“SELECT b FROM Book b”, Book.class).getResultList(); |
| 1 | 15:56:21,463 DEBUG [org.hibernate.SQL] – select book0\_.id as id1\_3\_, book0\_.publishingDate as publishi2\_3\_, book0\_.title as title3\_3\_, book0\_.version as version4\_3\_, book0\_.pages as pages1\_2\_ from Book book0\_ | |

### Hibernate has to use a similar approach for polymorphic queries. It has to left join the publication table with all tables of the subclasses, to get all Pubications of an Author.

|  |  |
| --- | --- |
| 123456789 | List authors= em.createQuery(“SELECT a FROM Author a”, Author.class).getResultList();for (Author a : authors) {    for (Publication p : a.getPublications()) {        if (p instanceof Book)        log(p.getTitle(), “book”);        else        log(p.getTitle(), “blog post”);    }} |
| 123 | 17:16:05,244 DEBUG [org.hibernate.SQL] – select author0\_.id as id1\_0\_, author0\_.firstName as firstNam2\_0\_, author0\_.lastName as lastName3\_0\_, author0\_.version as version4\_0\_ from Author author0\_17:16:05,280 DEBUG [org.hibernate.SQL] – select publicatio0\_.authorId as authorId2\_4\_0\_, publicatio0\_.publicationId as publicat1\_4\_0\_, publicatio1\_.id as id1\_3\_1\_, publicatio1\_.publishingDate as publishi2\_3\_1\_, publicatio1\_.title as title3\_3\_1\_, publicatio1\_.version as version4\_3\_1\_, publicatio1\_1\_.pages as pages1\_2\_1\_, publicatio1\_2\_.url as url1\_1\_1\_, case when publicatio1\_1\_.id is not null then 1 when publicatio1\_2\_.id is not null then 2 when publicatio1\_.id is not null then 0 end as clazz\_1\_ from PublicationAuthor publicatio0\_ inner join Publication publicatio1\_ on publicatio0\_.publicationId=publicatio1\_.id left outer join Book publicatio1\_1\_ on publicatio1\_.id=publicatio1\_1\_.id left outer join BlogPost publicatio1\_2\_ on publicatio1\_.id=publicatio1\_2\_.id where publicatio0\_.authorId=?Effective Java is a book. | |

## Choosing a Strategy

### Choosing the right inheritance strategy is not an easy task. As so often, you have to decide which advantages you need and which drawback you can accept for your application. Here are a few recommendations:

### If you require the best performance and need to use polymorphic queries and relationships, you should choose the single table strategy. But be aware, that you can’t use not null constraints on subclass attributes which increase the risk of data inconsistencies.

### If data consistency is more important than performance and you need polymorphic queries and relationships, the joined strategy is probably your best option.

### If you don’t need polymorphic queries or relationships, the table per class strategy is most likely the best fit. It allows you to use constraints to ensure data consistency and provides an option of polymorphic queries. But keep in mind, that polymorphic queries are very complex for this table structure and that you should avoid them.

In nutshell,

**MappedSuperclass** – the parent classes, can't be entities

**Single Table** – the entities from different classes with a common ancestor are placed in a single table

**Joined Table** – each class has its table and querying a subclass entity requires joining the tables

**Table-Per-Class** – all the properties of a class, are in its table, so no join is required

<https://www.dineshonjava.com/hibernate/implementing-inheritance-in-hibernate/>

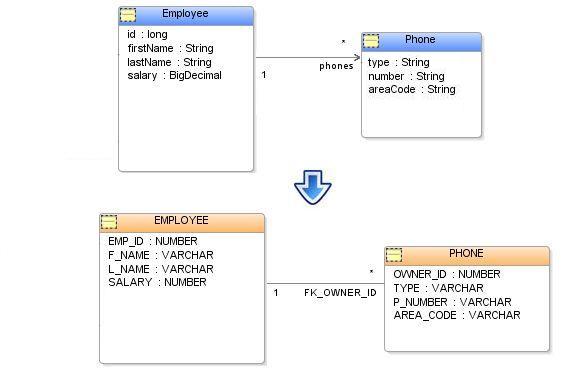
## Annotations used in Entity?

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| @Entity | This annotation specifies to declare the class as entity or a table. |
| @Table | This annotation specifies to declare table name. |
| @Basic | This annotation specifies non constraint fields explicitly. |
| @Embedded | This annotation specifies the properties of class or an entity whose value instance of an embeddable class. |
| @Id | This annotation specifies the property, use for identity (primary key of a table) of the class. |
| @GeneratedValue | This annotation specifies, how the identity attribute can be initialized such as Automatic, manual, or value taken from sequence table. |
| @Transient | This annotation specifies the property which in not persistent i.e. the value is never stored into database. |
| @Column | This annotation is used to specify column or attribute for persistence property. |
| @SequenceGenerator | This annotation is used to define the value for the property which is specified in @GeneratedValue annotation. It creates a sequence. |
| @TableGenerator | This annotation is used to specify the value generator for property specified in @GeneratedValue annotation. It creates a table for value generation. |
| @AccessType | This type of annotation is used to set the access type. If you set @AccessType(FIELD) then Field wise access will occur. If you set @AccessType(PROPERTY) then Property wise assess will occur. |
| @JoinColumn | This annotation is used to specify an entity association or entity collection. This is used in many- to-one and one-to-many associations. |
| @UniqueConstraint | This annotation is used to specify the field, unique constraint for primary or secondary table. |
| @ColumnResult | This annotation references the name of a column in the SQL query using select clause. |
| @ManyToMany | This annotation is used to define a many-to-many relationship between the join Tables. |
| @ManyToOne | This annotation is used to define a many-to-one relationship between the join Tables. |
| @OneToMany | This annotation is used to define a one-to-many relationship between the join Tables. |
| @OneToOne | This annotation is used to define a one-to-one relationship between the join Tables. |
| @NamedQueries | This annotation is used for specifying list of named queries. |
| @NamedQuery | This annotation is used for specifying a Query using static name. |

## Collection in JPA?

<https://en.wikibooks.org/wiki/Java_Persistence/ElementCollection>

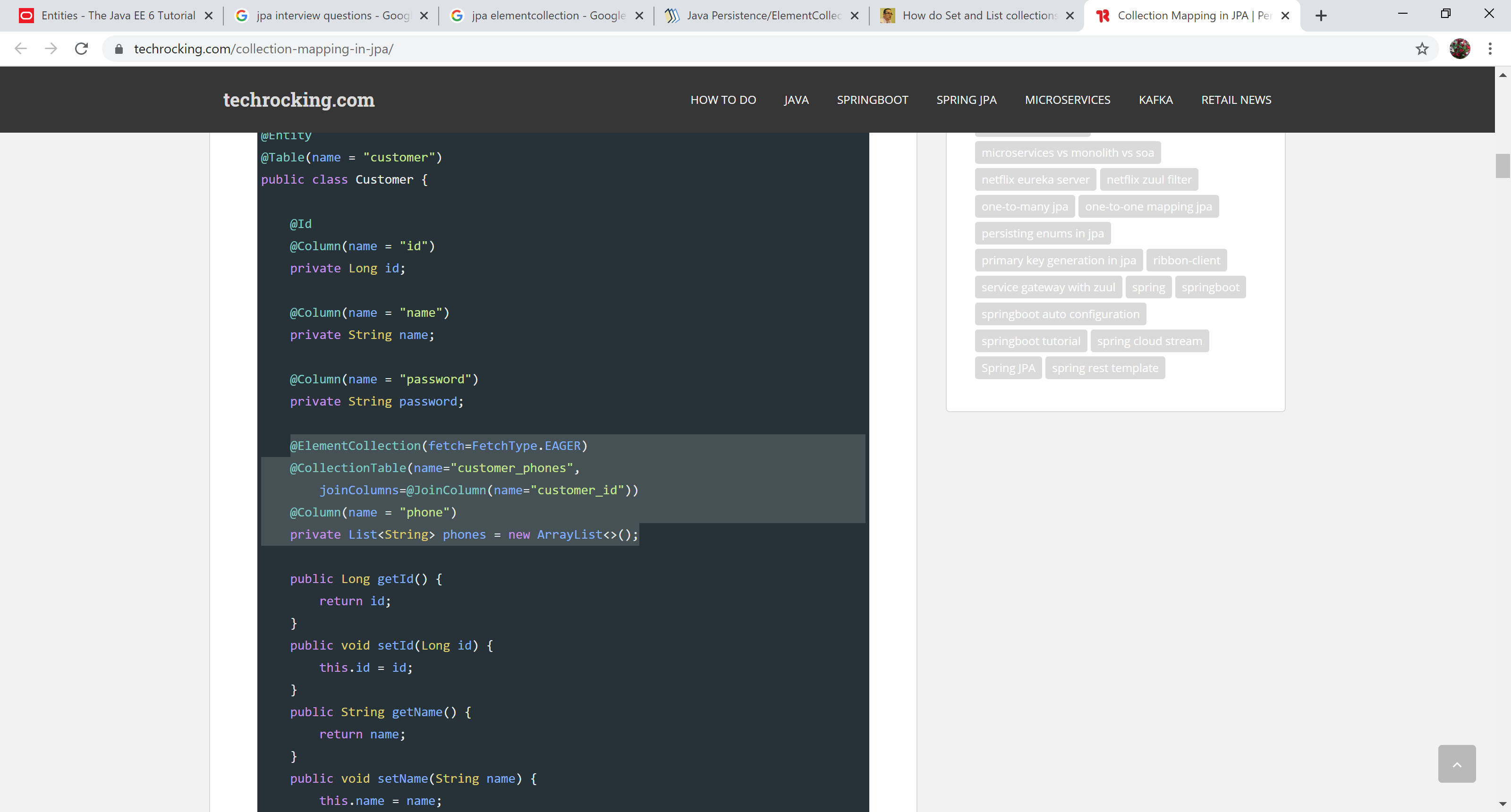
### JPA 2.0 defines an ElementCollection mapping. It is meant to handle several non-standard relationship mappings. An ElementCollection can be used to define a one-to-many relationship to an Embeddable object, or a Basic value (such as a collection of Strings). An ElementCollection can also be used in combination with a [Map](https://en.wikibooks.org/wiki/Java_Persistence/Relationships#Map_Key_Columns_.28JPA_2.0.29) to define relationships where the key can be any type of object, and the value is an Embeddable object or a Basic value.



### In JPA an ElementCollection relationship is defined through the [@ElementCollection](https://java.sun.com/javaee/6/docs/api/javax/persistence/ElementCollection.html) annotation or the <element-collection> element.

### The ElementCollection values are always stored in a separate table. The table is defined through the [@CollectionTable](https://java.sun.com/javaee/6/docs/api/javax/persistence/CollectionTable.html) annotation or the <collection-table> element. The CollectionTable defines the table's name and [@JoinColumn](https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinColumn.html) or [@JoinColumns](https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinColumns.html) if a composite primary key.

\*\*\* List, set and Map can be used as collection but as sper hibernate List is best.



<https://en.wikibooks.org/wiki/Java_Persistence/ElementCollection>

<https://techrocking.com/collection-mapping-in-jpa/>

## Difference between @OneToMany and @ElementCollection

### @ElementCollection allows you to simplify code when you want to implement one-to-many relationship with simple or embedded type. For instance in JPA 1.0 when you wanted to have a one-to-many relationship to a list of Strings, you had to create a simple entity POJO (StringWrapper) containing only primary key and the String in question:

@OneToMany

private Collection<StringWrapper> strings;

//...

public class StringWrapper {

@Id

private int id;

private String string;

}

With JPA 2.0 you can simply write:

@ElementCollection

private Collection<String> strings;

## Id generation types in JPA

### Following are the types of id generation strategy required to specify with @GeneratedValue annotation: -

### ***AUTO***: Hibernate selects the generation strategy based on the used dialect,

[@Id](https://www.objectdb.com/api/java/jpa/Id)

[@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue)([strategy](https://www.objectdb.com/api/java/jpa/GeneratedValue/strategy)=[GenerationType](https://www.objectdb.com/api/java/jpa/GenerationType).[AUTO](https://www.objectdb.com/api/java/jpa/GenerationType/AUTO))

long id;

### For most popular databases, it selects GenerationType.SEQUENCE

### ***IDENTITY***: Hibernate relies on an auto-incremented database column to generate the primary key.

### The [IDENTITY](https://www.objectdb.com/api/java/jpa/GenerationType/IDENTITY) strategy also generates an automatic value during commit for every new entity object. The difference is that a separate identity generator is managed per type hierarchy, so generated values are unique only per type hierarchy.

### The GenerationType.IDENTITY is the easiest to use but not the best one from a performance point of view. It relies on an auto-incremented database column and lets the database generate a new value with each insert operation. From a database point of view, this is very efficient because the auto-increment columns are highly optimized, and it doesn’t require any additional statements.

|  |  |
| --- | --- |
| 123 | @Id@GeneratedValue(strategy = GenerationType.IDENTITY)private Long id; |

### This approach has a significant drawback if you use Hibernate. Hibernate requires a primary key value for each managed entity and therefore has to perform the insert statement immediately. This prevents it from using [different optimization techniques](https://thorben-janssen.com/course-hibernate-performance-tuning/) like JDBC batching.

### ***SEQUENCE***: Hibernate requests the primary key value from a database sequence.

### The sequence strategy consists of two parts - defining a named sequence and using the named sequence in one or more fields in one or more classes. The [@SequenceGenerator](https://www.objectdb.com/api/java/jpa/SequenceGenerator) annotation is used to define a sequence and accepts a name, an initial value (the default is 1) and an allocation size (the default is 50).

### If you don’t provide any additional information, Hibernate will request the next value from its default sequence.

### A sequence is global to the application and can be used by one or more fields in one or more classes. The [SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE) strategy is used in the [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue) annotation to attach the given field to the previously defined named sequence:

### [@Entity](https://www.objectdb.com/api/java/jpa/Entity)

### // Define a sequence - might also be in another class:

### [@SequenceGenerator](https://www.objectdb.com/api/java/jpa/SequenceGenerator)([name](https://www.objectdb.com/api/java/jpa/SequenceGenerator/name)="seq", [initialValue](https://www.objectdb.com/api/java/jpa/SequenceGenerator/initialValue)=1, [allocationSize](https://www.objectdb.com/api/java/jpa/SequenceGenerator/allocationSize)=100)

### public class EntityWithSequenceId {

### // Use the sequence that is defined above:

### [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue)([strategy](https://www.objectdb.com/api/java/jpa/GeneratedValue/strategy)=[GenerationType](https://www.objectdb.com/api/java/jpa/GenerationType).[SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE), [generator](https://www.objectdb.com/api/java/jpa/GeneratedValue/generator)="seq")

### [@Id](https://www.objectdb.com/api/java/jpa/Id) long id;

### }

### Unlike [AUTO](https://www.objectdb.com/api/java/jpa/GenerationType/AUTO) and [IDENTITY](https://www.objectdb.com/api/java/jpa/GenerationType/IDENTITY), the [SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE) strategy generates an automatic value as soon as a new entity object is persisted (i.e. before commit). This may be useful when the primary key value is needed earlier. To minimize round trips to the database server, IDs are allocated in groups. The number of IDs in each allocation is specified by the [allocationSize](https://www.objectdb.com/api/java/jpa/SequenceGenerator/allocationSize) attribute. It is possible that some of the IDs in a given allocation will not be used. Therefore, this strategy does not guarantee there will be no gaps in sequence values.

### ***TABLE***: Hibernate uses a database table to simulate a sequence.

### The [TABLE](https://www.objectdb.com/api/java/jpa/GenerationType/TABLE) strategy is very similar to the [SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE) strategy:

### [@Entity](https://www.objectdb.com/api/java/jpa/Entity)

### [@TableGenerator](https://www.objectdb.com/api/java/jpa/TableGenerator)([name](https://www.objectdb.com/api/java/jpa/TableGenerator/name)="tab", [initialValue](https://www.objectdb.com/api/java/jpa/TableGenerator/initialValue)=0, [allocationSize](https://www.objectdb.com/api/java/jpa/TableGenerator/allocationSize)=50)

### public class EntityWithTableId {

### [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue)([strategy](https://www.objectdb.com/api/java/jpa/GeneratedValue/strategy)=[GenerationType](https://www.objectdb.com/api/java/jpa/GenerationType).[TABLE](https://www.objectdb.com/api/java/jpa/GenerationType/TABLE), [generator](https://www.objectdb.com/api/java/jpa/GeneratedValue/generator)="tab")

### [@Id](https://www.objectdb.com/api/java/jpa/Id) long id;

### }

### A tiny difference is related to the initial value attribute. Whereas the [SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE) strategy maintains the next sequence number to be used the [TABLE](https://www.objectdb.com/api/java/jpa/GenerationType/TABLE) strategy maintains the last value that was used. The implication for the initialValue attribute is that if you want sequence numbers to start with 1 in the [TABLE](https://www.objectdb.com/api/java/jpa/GenerationType/TABLE) strategy initialValue=0 has to be specified in the [@SequenceGenerator](https://www.objectdb.com/api/java/jpa/SequenceGenerator) annotation.

### The *GenerationType.TABLE* gets only rarely used nowadays. It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order. This slows down your application, and you should, therefore, prefer the *GenerationType.SEQUENCE*, if your database supports sequences, which most popular databases do.

### <https://www.baeldung.com/hibernate-identifiers>

<https://thorben-janssen.com/jpa-generate-primary-keys/>

<https://www.objectdb.com/java/jpa/entity/generated>

**Auto is default strategy.**

## Type of lock in JPA??

### JPA has six types of locks, list them in order of increasing reliability (from the most unreliable and fast, to the most reliable and slow):

### 1) NONE – without blocking

### 2) OPTIMISTIC (or READ synonym for JPA 1) – optimistic blocking

### 3) OPTIMISTIC\_FORCE\_INCREMENT (or WRITE synonym, remaining from JPA 1) – optimistic lock with forced increase of the version field,

### 4) PESSIMISTIC\_READ – pessimistic lock for reading,

### 5) PESSIMISTIC\_WRITE – pessimistic lock for write (and read),

### 6) PESSIMISTICI-PII-IZIETIHI, IHIETI , I, I, I, I, I, I, I, I; on record (and th with forced increase in the field of versioning.

## How can you change the fetch strategy settings of any Entity attributes for individual queries (query) or search methods (find), then if Enity has an attribute with fetchType = LAZY, but for a specific query you need to make it EAGER or vice versa?

### For this, the EntityGraph API exists, it is used like this: using the NamedEntityGraph annotation for an Entity, it creates named EntityGraph objects that contain a list of attributes that need to change fetchType to EAGER, and then this name is specified in hits queries or the find method. As a result, the fetchType attribute of the Entity changes, but only for this request. There are two standard properties for specifying EntityGraph in hit:

### 1) javax.persistence.fetchgraph – all attributes listed in EntityGraph change fetchType to EAGER, all others to LAZY

### 2) javax.persistence.loadgraph – all attributes listed in EntityGraph change fetchType to EAGER, all others retain their fetchType (that is, if the attribute not specified in EntityGraph has fetchType was EAGER, then it will remain EAGER) With NamedSubgraph you can also change fetchType nested Entity objects.

## @Transactional

# JPQL

## What is JPQL and why should we use this?

### The Java Persistence Query language (JPQL) is a part of JPA specification that defines searches against persistence entities. It is an object-oriented query language which is used to perform database operations on persistent entities. Instead of the database table, JPQL uses entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks. JPQL is an extension of Entity JavaBeans Query Language (EJBQL).

### Some of the essential features of JPQL are: -

### It is simple and robust.

### It is a platform-independent query language.

### JPQL queries can be declared statically into metadata or can also be dynamically built in code.

### It can be used with any database such as MySQL, Oracle.

<https://www.objectdb.com/java/jpa/query/jpql/collection>

# CRITERIA API

## What is criteria api?

### The Criteria API is a specification that provides type-safe and portable criteria queries written using Java programming language APIs. It is one of the most common ways of constructing queries for entities and their persistent state. It is just an alternative method for defining JPA queries. Criteria API defines a platform-independent criteria queries, written in Java programming language. It was introduced in JPA 2.0. The main purpose behind this is to provide a type-safe way to express a query.

# Cache

## Types of cache in JPa?

### JPA talks about two kinds of caches (cache):

### 1) first-level cache (first-level cache) —caches data from a single transaction; Data is saved in transaction. Scope is transacational and hence data is not shared between wo transactions.

### 2) second-level cache (second-level cache) —caches data for more than one transaction. The JPA provider can, but is not required to implement work with the second-level cache. This kind of cache can save access time and improve performance, but the downside is the ability to get outdated data.

### As most other fully-equipped ORM frameworks, Hibernate has the concept of **first-level cache. It is a session scoped cache** which ensures that each entity instance is loaded only once in the persistent context.

### Once the session is closed, first-level cache is terminated as well. This is actually desirable, as it allows for concurrent sessions to work with entity instances in isolation from each other.

### On the other hand, **second-level cache is *SessionFactory*-scoped**, meaning it is shared by all sessions created with the same session factory. When an entity instance is looked up by its id (either by application logic or by Hibernate internally, *e.g.* when it loads associations to that entity from other entities), and if second-level caching is enabled for that entity, the following happens:

### If an instance is already present in the first-level cache, it is returned from there If an instance is not found in the first-level cache, and the corresponding instance state is cached in the second-level cache, then the data is fetched from there and an instance is assembled and returned

### Otherwise, the necessary data are loaded from the database and an instance is assembled and returned

### Once the instance is stored in the persistence context (first-level cache), it is returned from there in all subsequent calls within the same session until the session is closed or the instance is manually evicted from the persistence context. Also, the loaded instance state is stored in L2 cache if it was not there already.

## How to enable second level cache in hibernate?

1. **Add entry in POM**

### <dependency>

### <groupId>org.hibernate</groupId>

### <artifactId>hibernate-ehcache</artifactId>

### <version>5.2.2.Final</version>

### </dependency>

### **Add property in application.proerties**

### hibernate.cache.use\_second\_level\_cache=true

### hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

### **Annotate entities with @Cacheble**

### @Entity

@Cacheable

@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)

public class Foo {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "ID")

private long id;

@Column(name = "NAME")

private String name;

// getters and setters

}

<https://www.baeldung.com/hibernate-second-level-cache>

## What are the options for setting the second-level cache (second-level cache) in JPA or what similarly describe what values ​​can the shared-cache-mode element take from persistence.xml?

### 1) ALL – all Entity can be cached in the second-level cache ,

### 2) NONE – caching is disabled for all Entity,

### 3) ENABLE\_SELECTIVE — caching only works for those Entities that have the Cacheable (true) or their xml equivalent set, for all others, caching is disabled,

### 4) DISABLE\_SELECTIVE — caching works for all Entity, except for those that have the Cacheable annotation (false) or its xml equivalent is

### 5) UNSPECIFIED – caching is not defined, each provider EPA uses its default value for caching,

## Cache API to delete object from cache?

<https://en.wikibooks.org/wiki/Java_Persistence/Caching>

### To work with the second level cache (second level cache) in JPA, the Cache interface is described which contains a large number of methods for managing the second level cache (second level cache), if supported by the JPA provider, of course. The object of this interface can be retrieved using the getCache method from EntityManagerFactory.

## How to make entity cacheable?

### Cacheable – allows you to enable or disable the use of a second-level cache (second-level cache) for this Entity (if the JPA provider supports caching and cache settings (second-level cache) are ENABLE\_SELECTIVE or DISABLE\_SELECTIVE,). Note that the property is inherited and if it is not blocked by the heirs, then caching will change for them too

# Hibernate only features

# REFERENCES

<https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html>

Udemy course

<https://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/>

<https://en.wikibooks.org/wiki/Java_Persistence/Basic_Attributes>

<https://www.objectdb.com/java/jpa/persistence/delete>

<https://vladmihalcea.com/orphanremoval-jpa-hibernate/>

<https://www.infoworld.com/article/3373652/java-persistence-with-jpa-and-hibernate-part-1-entities-and-relationships.html?page=3>