# Hibernate:

Hibernate provides implementation of Java Persistence API, so we can use JPA annotations with model beans and hibernate will take care of configuring it to be used in CRUD operations. We will look into this with annotations example.

#### Explain all annotations which you used in Hibernate.

@Entity (javax.persistence.Entity)

To specify to hibernate engine that decorated class is Entity and has same mapped table in db.

@Entity

public class Company implements Serializable {

...

}

@Table( javax.persistence.Table)

Decorated class is specified with table mapping.

@Entity

@Table(name = "company")

public class Company implements Serializable {

...

}

@Column( javax.persistence.Column)

To specify column mapping in db table

@Entity

@Table(name = "company")

public class Company implements Serializable {

  @Column(name = "name")

  private String name;

...

}

@Id javax.persistence.Id;

Specified field is id in table of db.

@Entity

@Table(name = "company")

public class Company implements Serializable {

  @Id

  @Column(name = "id")

  private int id;

...

}

@GeneratedValue javax.persistence.GeneratedValue;

Specified field is auto generated value.

@Entity

@Table(name = "company")

public class Company implements Serializable {

  @Id

  @Column(name = "id")

  @GeneratedValue

  private int id;

...

}

@Version javax.persistence.Version;

@Entity

@Table(name = "company")

public class Company implements Serializable {

  @Version

  @Column(name = "version")

  private Date version;

...

}

@OrderBy javax.persistence.OrderBy;

@OrderBy("firstName asc")

private Set contacts;

@Transient javax.persistence.Transient;

Annotate your transient properties with @Transient.

@Lob javax.persistence.Lob;

Annotate large objects with @Lob

###### Hibernate Association Mapping Annotations

@OneToOne javax.persistence.OneToOne;

@ManyToOne javax.persistence.ManyToOne;

@OneToMany javax.persistence.OneToMany;

@ManyToMany javax.persistence.ManyToMany;

@PrimaryKeyJoinColumn javax.persistence.PrimaryKeyJoinColumn;

@JoinColumn javax.persistence.JoinColumn;

@JoinTable javax.persistence.JoinTable;

@MapsId javax.persistence.MapsId;

###### Hibernate Inheritance Mapping Annotations

@Inheritance javax.persistence.Inheritance;

@DiscriminatorColumn javax.persistence.DiscriminatorColumn;

@DiscriminatorValue javax.persistence.DiscriminatorValue;

#### What is other option if I don’t want to use @Generated annotation in Hibernate entity class…

JPA offers 4 different ways to generate primary key values:

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

The GenerationType.AUTO is the default generation type and lets the persistence provider choose the generation strategy.

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

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

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.

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id", updatable = false, nullable = false)

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://www.thoughts-on-java.org/course-hibernate-performance-tuning/) like JDBC batching.

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

It requires additional select statements to get the next value from a database sequence. But this has no performance impact for most applications. And if your application has to persist a huge number of new entities, you can use some Hibernate specific optimizations to reduce the number of statements.

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

If you don’t provide any additional information, Hibernate will request the next value from its default sequence. You can change that by referencing the name of a @SequenceGenerator in the generator attribute of the @GeneratedValue annotation. The @SequenceGenerator annotation lets you define the name of the generator, the name, and schema of the database sequence and the allocation size of the sequence.

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book\_generator")

@SequenceGenerator(name="book\_generator", sequenceName = "book\_seq", allocationSize=50)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

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

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.

@Id

@GeneratedValue(strategy = GenerationType.TABLE)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

*You can use the @TableGenerator annotation in a similar way as the already explained @SequenceGenerator annotation to specify the database table which Hibernate shall use to simulate the sequence.*

@Id

@GeneratedValue(strategy = GenerationType.TABLE, generator = "book\_generator")

@TableGenerator(name="book\_generator", table="id\_generator", schema="bookstore")

@Column(name = "id", updatable = false, nullable = false)

private Long id;

#### What do you understand by lazy and eager fetch type in hibernate.

Department (1) ----------------(\*) Employee

In Department Object field collection of Employee is present as association

EAGER Loading can be simply enabled using the following annotation parameter:

fetch = FetchType.EAGER

**package** com.kc.hib.entity;  
  
**import** javax.persistence.\*;  
**import** java.io.Serializable;  
**import** java.util.HashSet;  
**import** java.util.Set;  
  
*/\*\*  
 \* Created by keshav.chaure on 5/31/2018.  
 \*/*@Entity  
@Table(name = **"Department"**)  
**public class** Department **implements** Serializable {  
  
 **private int deptId**;  
 **private** String **deptName**;  
 **private** String **deptCode**;  
 Set<Employee> **employees** = **new** HashSet<>(0);  
  
 **public** Department() {  
 System.***out***.println(**"dept loading.."**);  
 }  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***) *// use autoincreament* @Column(name = **"dept\_id"**)  
 **public int** getDeptId() {  
 **return deptId**;  
 }  
  
 **public void** setDeptId(**int** deptId) {  
 **this**.**deptId** = deptId;  
 }  
  
 @Column(name = **"dept\_name"**)  
 **public** String getDeptName() {  
 **return deptName**;  
 }  
  
 **public void** setDeptName(String deptName) {  
 **this**.**deptName** = deptName;  
 }  
  
  
 @Column(name = **"dept\_code"**)  
 **public** String getDeptCode() {  
 **return deptCode**;  
 }  
  
 **public void** setDeptCode(String deptCode) {  
 **this**.**deptCode** = deptCode;  
 }  
  
 @OneToMany(mappedBy = **"dept"**,fetch = FetchType.***EAGER***)  
 **public** Set<Employee> getEmployees() {  
 **return employees**;  
 }  
  
 **public void** setEmployees(Set<Employee> employees) {  
 **this**.**employees** = employees;  
 }  
}

**package** com.kc.hib.entity;  
  
**import** javax.persistence.\*;  
**import** java.io.Serializable;  
  
*/\*\*  
 \* Created by keshav.chaure on 5/31/2018.  
 \*/*@Entity  
@Table(name=**"Employee"**)  
**public class** Employee **implements** Serializable{  
  
  
 **private int empId**;  
 **private** String **empName**;  
 **private** String **empCity**;  
 **private** Department **dept**;  
  
 **public** Employee() {  
 System.***out***.println(**"employee loading.."**);  
 }  
  
 **public** Employee(String empName, String empCity, Department dept) {  
 **this**.**empName** = empName;  
 **this**.**empCity** = empCity;  
 **this**.**dept** = dept;  
 }  
  
 @Id  
 @GeneratedValue  
 @Column(name=**"emp\_id"**)  
 **public int** getEmpId() {  
 **return empId**;  
 }  
  
 **public void** setEmpId(**int** empId) {  
 **this**.**empId** = empId;  
 }  
  
 @Column(name=**"emp\_name"**)  
 **public** String getEmpName() {  
 **return empName**;  
 }  
  
 **public void** setEmpName(String empName) {  
 **this**.**empName** = empName;  
 }  
  
 @Column(name=**"emp\_city"**)  
 **public** String getEmpCity() {  
 **return empCity**;  
 }  
  
 **public void** setEmpCity(String empCity) {  
 **this**.**empCity** = empCity;  
 }  
  
 @ManyToOne(fetch = FetchType.***EAGER***)  
 @JoinColumn(name = **"emp\_dept"**)  
 **public** Department getDept() {  
 **return dept**;  
 }  
  
 **public void** setDept(Department dept) {  
 **this**.**dept** = dept;  
 }  
}

To use LAZY Fetching the following parameter is used:

fetch = FetchType.LAZY

**package** com.kc.hib.entity;  
  
**import** javax.persistence.\*;  
**import** java.io.Serializable;  
**import** java.util.HashSet;  
**import** java.util.Set;  
  
*/\*\*  
 \* Created by keshav.chaure on 5/31/2018.  
 \*/*@Entity  
@Table(name = **"Department"**)  
**public class** Department **implements** Serializable {  
  
 **private int deptId**;  
 **private** String **deptName**;  
 **private** String **deptCode**;  
 *// @JoinTable(name = "Dept\_Emp", joinColumns = { @JoinColumn(name = "dept\_id") }, inverseJoinColumns = { @JoinColumn(name = "emp\_id") })* Set<Employee> **employees** = **new** HashSet<>(0);  
  
 **public** Department() {  
 System.***out***.println(**"dept loading.."**);  
 }  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***) *// use autoincreament  
 // @GeneratedValue(strategy = GenerationType.AUTO) // default genrater strategy it will genrate next value as per database dialect.  
 // @GeneratedValue(strategy = GenerationType.SEQUENCE) org.hibernate.MappingExcepton : org.hibernate.dialect.MySQLDialect does not support sequence.  
 // @GeneratedValue(strategy = GenerationType.TABLE)* @Column(name = **"dept\_id"**)  
 **public int** getDeptId() {  
 **return deptId**;  
 }  
  
 **public void** setDeptId(**int** deptId) {  
 **this**.**deptId** = deptId;  
 }  
  
 @Column(name = **"dept\_name"**)  
 **public** String getDeptName() {  
 **return deptName**;  
 }  
  
 **public void** setDeptName(String deptName) {  
 **this**.**deptName** = deptName;  
 }  
  
  
 @Column(name = **"dept\_code"**)  
 **public** String getDeptCode() {  
 **return deptCode**;  
 }  
  
 **public void** setDeptCode(String deptCode) {  
 **this**.**deptCode** = deptCode;  
 }  
  
 @OneToMany(mappedBy = **"dept"**,fetch = FetchType.***LAZY***)  
 **public** Set<Employee> getEmployees() {  
 **return employees**;  
 }  
  
 **public void** setEmployees(Set<Employee> employees) {  
 **this**.**employees** = employees;  
 }  
}

**package** com.kc.hib.entity;  
  
**import** javax.persistence.\*;  
**import** java.io.Serializable;  
  
*/\*\*  
 \* Created by keshav.chaure on 5/31/2018.  
 \*/*@Entity  
@Table(name=**"Employee"**)  
**public class** Employee **implements** Serializable{  
  
  
 **private int empId**;  
 **private** String **empName**;  
 **private** String **empCity**;  
 **private** Department **dept**;  
  
 **public** Employee() {  
 System.***out***.println(**"employee loading.."**);  
 }  
  
 **public** Employee(String empName, String empCity, Department dept) {  
 **this**.**empName** = empName;  
 **this**.**empCity** = empCity;  
 **this**.**dept** = dept;  
 }  
  
 @Id  
 @GeneratedValue  
 @Column(name=**"emp\_id"**)  
 **public int** getEmpId() {  
 **return empId**;  
 }  
  
 **public void** setEmpId(**int** empId) {  
 **this**.**empId** = empId;  
 }  
  
 @Column(name=**"emp\_name"**)  
 **public** String getEmpName() {  
 **return empName**;  
 }  
  
 **public void** setEmpName(String empName) {  
 **this**.**empName** = empName;  
 }  
  
 @Column(name=**"emp\_city"**)  
 **public** String getEmpCity() {  
 **return empCity**;  
 }  
  
 **public void** setEmpCity(String empCity) {  
 **this**.**empCity** = empCity;  
 }  
  
 @ManyToOne(fetch = FetchType.***LAZY***)  
 @JoinColumn(name = **"emp\_dept"**)  
 **public** Department getDept() {  
 **return dept**;  
 }  
  
 **public void** setDept(Department dept) {  
 **this**.**dept** = dept;  
 }  
}

In the next section we will look at the differences between the two types of fetching.

Lazy Loading

Advantages:

Initial load time much smaller than in the other approach

Less memory consumption than in the other approach

Disadvantages:

Delayed initialization might impact performance during unwanted moments

In some cases you need to handle lazily-initialized objects with a special care or you might end up with an exception

Eager Loading:

Advantages:

No delayed initialization related performance impacts

Disadvantages:

Long initial loading time

Loading too much unnecessary data might impact performance

Hibernate applies lazy loading approach on entities and associations by providing a proxy implementation of classes.

Hibernate intercepts calls to an entity by substituting it with a proxy derived from an entity’s class. In our example, when a requested information is missing, it will be loaded from a database before control is ceded to the User class implementation.

It should also be noted that when the association is represented as a collection class (in the above examples it is represented as Set<Employee> employees), then a wrapper is created and substituted for an original collection.

Let’s assume that you’re writing code that’d track the price of mobile phones. Now, let’s say you have a collection of objects representing different Mobile phone vendors (MobileVendor), and each vendor has a collection of objects representing the PhoneModels they offer.

To put it simple, there’s exists a one-to-many relationship between MobileVendor:PhoneModel.

**MobileVendor Class**

Class MobileVendor{

long vendor\_id;

PhoneModel[] phoneModels;

...

}

Okay, so you want to print out all the details of phone models. A naive O/R implementation would SELECT all mobile vendors and then do N additional SELECTs for getting the information of PhoneModel for each vendor.

-- Get all Mobile Vendors

SELECT \* FROM MobileVendor;

-- For each MobileVendor, get PhoneModel details

SELECT \* FROM PhoneModel WHERE MobileVendor.vendorId=?

As you see, the N+1 problem can happen if the first query populates the primary object and the second query populates all the child objects for each of the unique primary objects returned.

**Resolve N+1 SELECTs problem**

**(i) HQL fetch join**

"from MobileVendor mobileVendor join fetch mobileVendor.phoneModel PhoneModels"

Corresponding SQL would be (assuming tables as follows: t\_mobile\_vendor for MobileVendor and t\_phone\_model for PhoneModel)

SELECT \* FROM t\_mobile\_vendor vendor LEFT OUTER JOIN t\_phone\_model model ON model.vendor\_id=vendor.vendor\_id

**(ii) Criteria query**

Criteria criteria = session.createCriteria(MobileVendor.class);

criteria.setFetchMode("phoneModels", FetchMode.EAGER);

In both cases, our query returns a list of MobileVendor objects with the phoneModels initialized. Only one query needs to be run to return all the PhoneModel and MobileVendor information required.

#### Fetching strategies in hibernate OR Hibernate n+1 problem

Fetching Strategies

Hibernate uses a fetching strategy to retrieve associated objects if the application needs to navigate the association. Fetch strategies can be declared in the O/R mapping metadata, or over-ridden by a particular HQL or Criteria query.

Hibernate defines the following fetching strategies:

**Join fetching**: Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.

**Select fetching**: a second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.

**Subselect fetching**: a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.

**Batch fetching:** an optimization strategy for select fetching. Hibernate retrieves a batch of entity instances or collections in a single SELECT by specifying a list of primary or foreign keys.

There are four fetching strategies

**Hibernate also distinguishes between:**

Immediate fetching: an association, collection or attribute is fetched immediately when the owner is loaded.

Lazy collection fetching: a collection is fetched when the application invokes an operation upon that collection. This is the default for collections.

"Extra-lazy" collection fetching: individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed. It is suitable for large collections.

Proxy fetching: a single-valued association is fetched when a method other than the identifier getter is invoked upon the associated object.

"No-proxy" fetching: a single-valued association is fetched when the instance variable is accessed. Compared to proxy fetching, this approach is less lazy; the association is fetched even when only the identifier is accessed. It is also more transparent, since no proxy is visible to the application. This approach requires buildtime bytecode instrumentation and is rarely necessary.

Lazy attribute fetching: an attribute or single valued association is fetched when the instance variable is accessed. This approach requires buildtime bytecode instrumentation and is rarely necessary.

We have two orthogonal notions here: when is the association fetched and how is it fetched. It is important that you do not confuse them. We use fetch to tune performance. We can use lazy to define a contract for what data is always available in any detached instance of a particular class.

1. fetch-“join” = Disable the lazy loading, always load all the collections and entities.

2. fetch-“select” (default) = Lazy load all the collections and entities.

3. batch-size=”N” = Fetching up to ‘N’ collections or entities, \*Not record\*.

4. fetch-“subselect” = Group its collection into a sub select statement.

Here’s a “one-to-many relationship” example for the fetching strategies demonstration. A stock is belong to many stock daily records.

Example to declare fetch strategies in XML file

...

<hibernate-mapping>

<class name="com.mkyong.common.Stock" table="stock">

<set name="stockDailyRecords" cascade="all" inverse="true"

table="stock\_daily\_record" batch-size="10" fetch="select">

<key>

<column name="STOCK\_ID" not-null="true" />

</key>

<one-to-many class="com.mkyong.common.StockDailyRecord" />

</set>

</class>

</hibernate-mapping>

Copy

Example to declare fetch strategies in annotation

...

@Entity

@Table(name = "Department", catalog = "keshav\_db")

public class Department implements Serializable{

...

@OneToMany(fetch = FetchType.LAZY, mappedBy = "dept")

@Cascade(CascadeType.ALL)

@Fetch(FetchMode.SELECT)

@BatchSize(size = 10)

public Set<Employee> getEmployee() {

return this.employees;

}

...

}

Copy

Let explore how fetch strategies affect the Hibernate generated SQL statement.

##### fetch=”select” or @Fetch(FetchMode.SELECT)

This is the default fetching strategy. it enabled the lazy loading of all it’s related collections. Let see the example…

//call select from Department

Department d=(Department)session.get(Department.class,2);

Set<Employee> sets=d.getEmployees();

//call select from Employee

for (Employee e:sets) {  
 System.out.println(e.getEmpId());  
}

Output

Hibernate:

Hibernate:

select

department0\_.dept\_id as dept\_id1\_0\_0\_,

department0\_.dept\_code as dept\_cod2\_0\_0\_,

department0\_.dept\_name as dept\_nam3\_0\_0\_

from

Department department0\_

where

department0\_.dept\_id=?

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_dept as emp\_dept4\_1\_1\_,

employees0\_.emp\_city as emp\_city2\_1\_1\_,

employees0\_.emp\_name as emp\_name3\_1\_1\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

Hibernate generated two select statements

1. Select statement to retrieve the Department records –session.get(Department.class, 1)

2. Select its related collections – sets.iterator()

##### fetch=”join” or @Fetch(FetchMode.JOIN)

The “join” fetching strategy will disabled the lazy loading of all it’s related collections. Let see the example…

// call select from Department and Employee

Department d=(Department)session.get(Department.class,2);

Set<Employee> sets=d.getEmployees();

//No extra select herer

for (Employee e:sets) {  
 System.out.println(e.getEmpId());  
}

Output

Hibernate:

Hibernate:

select

department0\_.dept\_id as dept\_id1\_0\_0\_,

department0\_.dept\_code as dept\_cod2\_0\_0\_,

department0\_.dept\_name as dept\_nam3\_0\_0\_,

employees1\_.emp\_dept as emp\_dept4\_0\_1\_,

employees1\_.emp\_id as emp\_id1\_1\_1\_,

employees1\_.emp\_id as emp\_id1\_1\_2\_,

employees1\_.emp\_dept as emp\_dept4\_1\_2\_,

employees1\_.emp\_city as emp\_city2\_1\_2\_,

employees1\_.emp\_name as emp\_name3\_1\_2\_

from

Department department0\_

left outer join

Employee employees1\_

on department0\_.dept\_id=employees1\_.emp\_dept

where

department0\_.dept\_id=?

Hibernate generated only one select statement, it retrieve all its related collections when the Department is initialized. –session.get(Department.class, 1)

Select statement to retrieve the Department records and outer join its related collections.

##### batch-size=”10″ or @BatchSize(size = 10)

This ‘batch size’ fetching strategy is always misunderstanding by many Hibernate developers. Let see the \*misunderstand\* concept here…

Department d=(Department)session.get(Department.class,2);

Set<Employee> sets=d.getEmployees();

for (Employee e:sets) {  
 System.out.println(e.getEmpId());  
}

What is your expected result, is this per-fetch 10 records from collection? See the output

Output

Hibernate:

select

department0\_.dept\_id as dept\_id1\_0\_0\_,

department0\_.dept\_code as dept\_cod2\_0\_0\_,

department0\_.dept\_name as dept\_nam3\_0\_0\_

from

Department department0\_

where

department0\_.dept\_id=?

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_dept as emp\_dept4\_1\_0\_,

employees0\_.emp\_city as emp\_city2\_1\_0\_,

employees0\_.emp\_name as emp\_name3\_1\_0\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

The batch-size did nothing here, it is not how batch-size work. See this statement.

The batch-size fetching strategy is not define how many records inside in the collections are loaded. Instead, it defines how many collections should be loaded.

— Repeat N times until you remember this statement —

Another example

Let see another example, you want to print out all the department records and its related employee (collections) one by one.

List<Department> list = session.createQuery("from Department").list();  
  
for(Department department : list){  
  
 Set employeeSet = department.getEmployees();  
  
 for (Iterator iter = employeeSet.iterator(); iter.hasNext(); ) {  
 Employee emp = (Employee) iter.next();  
 System.out.println(emp.getEmpId());  
 System.out.println(emp.getEmpName());  
 }  
}

###### No batch-size fetching strategy

Output

Hibernate:

select

department0\_.dept\_id as dept\_id1\_0\_,

department0\_.dept\_code as dept\_cod2\_0\_,

department0\_.dept\_name as dept\_nam3\_0\_

from

Department department0\_

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_dept as emp\_dept4\_1\_1\_,

employees0\_.emp\_city as emp\_city2\_1\_1\_,

employees0\_.emp\_name as emp\_name3\_1\_1\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_dept as emp\_dept4\_1\_1\_,

employees0\_.emp\_city as emp\_city2\_1\_1\_,

employees0\_.emp\_name as emp\_name3\_1\_1\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_dept as emp\_dept4\_1\_1\_,

employees0\_.emp\_city as emp\_city2\_1\_1\_,

employees0\_.emp\_name as emp\_name3\_1\_1\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_dept as emp\_dept4\_1\_1\_,

employees0\_.emp\_city as emp\_city2\_1\_1\_,

employees0\_.emp\_name as emp\_name3\_1\_1\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_dept as emp\_dept4\_1\_1\_,

employees0\_.emp\_city as emp\_city2\_1\_1\_,

employees0\_.emp\_name as emp\_name3\_1\_1\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

Keep repeat the select statements....depend how many departments records in your table.

Copy

If you have 5 department records in the database, the Hibernate’s default fetching strategies will generate 5 +1 select statements and hit the database.

1. Select statement to retrieve all the Department records.

2. Select its **1st** collection

…..

N . Select its **nth** collection

The generated queries are not efficient and caused a serious performance issue.

###### Enabled the batch-size=’10’ fetching strategy

Let see another example with batch-size=’10’ is enabled.

Output

Hibernate:

select

department0\_.dept\_id as dept\_id1\_0\_,

department0\_.dept\_code as dept\_cod2\_0\_,

department0\_.dept\_name as dept\_nam3\_0\_

from

Department department0\_

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_dept as emp\_dept4\_1\_0\_,

employees0\_.emp\_city as emp\_city2\_1\_0\_,

employees0\_.emp\_name as emp\_name3\_1\_0\_

from

Employee employees0\_

where

employees0\_.emp\_dept in (

?, ?

)

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_dept as emp\_dept4\_1\_0\_,

employees0\_.emp\_city as emp\_city2\_1\_0\_,

employees0\_.emp\_name as emp\_name3\_1\_0\_

from

Employee employees0\_

where

employees0\_.emp\_dept in ( ?, ? )

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_dept as emp\_dept4\_1\_0\_,

employees0\_.emp\_city as emp\_city2\_1\_0\_,

employees0\_.emp\_name as emp\_name3\_1\_0\_

from

Employee employees0\_

where

employees0\_.emp\_dept=?

Now, Hibernate will per-fetch the collections, with a select \*in\* statement. If you have 6 Employee records, it will generate 4 select statements.

1. Select statement to retrieve all the Stock records.

2. Select In statement to per-fetch its related collections (2 collections a time)

3. Select In statement to per-fetch its related collections (next 2 collections a time)

4. Select In statement to per-fetch its related collections (next 2 collections a time)

With batch-size enabled, it simplify the select statements from 21 select statements to 3 select statements.

##### fetch=”subselect” or @Fetch(FetchMode.SUBSELECT)

This fetching strategy is enable all its related collection in a sub select statement. Let see the same query again..

List<Department> list = session.createQuery("from Department").list();  
  
for(Department department : list){  
  
 Set employeeSet = department.getEmployees();  
  
 for (Iterator iter = employeeSet.iterator(); iter.hasNext(); ) {  
 Employee emp = (Employee) iter.next();  
 System.out.println(emp.getEmpId());  
 System.out.println(emp.getEmpName());  
 }  
}

Output

Hibernate:

select department0\_.dept\_id as dept\_id1\_0\_,

department0\_.dept\_code as dept\_cod2\_0\_,

department0\_.dept\_name as dept\_nam3\_0\_

from

Department department0\_

Hibernate:

select

employees0\_.emp\_dept as emp\_dept4\_0\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_1\_,

employees0\_.emp\_id as emp\_id1\_1\_0\_,

employees0\_.emp\_dept as emp\_dept4\_1\_0\_,

employees0\_.emp\_city as emp\_city2\_1\_0\_,

employees0\_.emp\_name as emp\_name3\_1\_0\_

from

Employee employees0\_

where

employees0\_.emp\_dept in (

select

department0\_.dept\_id

from

Department department0\_

)

With “subselect” enabled, it will create two select statements.

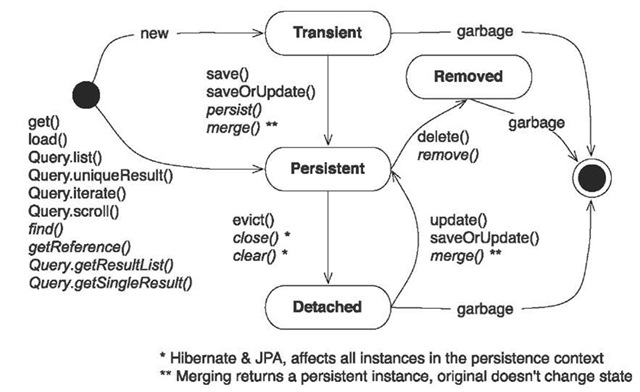
1. Select statement to retrieve all the Stock records.

2. Select all its related collections in a sub select query.

Conclusion

The fetching strategies are highly flexible and a very important tweak to optimize the Hibernate query, but if you used it in a wrong place, it will be a total disaster.

#### The entity states in hibernate



**Transient objects**

Objects instantiated using the new operator aren’t immediately persistent. Their state is transient, which means they aren’t associated with any database table row and so their state is lost as soon as they’re no longer referenced by any other object. These objects have a lifespan that effectively ends at that time, and they become inaccessible and available for garbage collection. Java Persistence doesn’t include a term for this state; entity objects you just instantiated are new. We’ll continue to refer to them as transient to emphasize the potential for these instances to become managed by a persistence service.

Hibernate and Java Persistence consider all transient instances to be nontransactional; any modification of a transient instance isn’t known to a persistence context. This means that Hibernate doesn’t provide any roll-back functionality for transient objects.

Objects that are referenced only by other transient instances are, by default, also transient. For an instance to transition from transient to persistent state, to become managed, requires either a call to the persistence manager or the creation of a reference from an already persistent instance.

**Persistent objects**

A persistent instance is an entity instance with a database identity, as defined. “Mapping entities with identity.” That means a persistent and managed instance has a primary key value set as its database identifier. (There are some variations to when this identifier is assigned to a persistent instance.)

Persistent instances may be objects instantiated by the application and then made persistent by calling one of the methods on the persistence manager. They may even be objects that became persistent when a reference was created from another persistent object that is already managed. Alternatively, a persistent instance may be an instance retrieved from the database by execution of a query, by an identifier lookup, or by navigating the object graph starting from another persistent instance.

Persistent instances are always associated with a persistence context. Hibernate caches them and can detect whether they have been modified by the application.

There is much more to be said about this state and how an instance is managed in a persistence context. We’ll get back to this later in this topic.

**Removed objects**

You can delete an entity instance in several ways: For example, you can remove it with an explicit operation of the persistence manager. It may also become available for deletion if you remove all references to it, a feature available only in Hibernate or in Java Persistence with a Hibernate extension setting (orphan deletion for entities).

An object is in the removed state if it has been scheduled for deletion at the end of a unit of work, but it’s still managed by the persistence context until the unit of work completes. In other words, a removed object shouldn’t be reused because it will be deleted from the database as soon as the unit of work completes. You should also discard any references you may hold to it in the application (of course, after you finish working with it—for example, after you’ve rendered the removal-confirmation screen your users see).

#### many-to-many syntax in hibernate

#### explain hibernate dialect property

#### what is minimum requirement for ORM to connect to any database

#### What is transaction ? is it necessary to use for executing one sql .

A Transaction is a unit of work in which all the operations must be executed or none of them. To understand the importance of transaction, think of an example which applies on all of us i.e. Transferring Amount from one account to another as this operation includes the below two steps:

1.Deduct the balance from the sender’s bank account

2.Add the amount to the receiver’s bank account

Now think a situation where the amount is deducted from sender’s account but is not delivered to receiver’s account due to some errors. Such issues are managed by the transaction management where both steps are performed in a single unit. In the case of a failure, the transaction should be roll-backed.

Database, or system, transaction boundaries are always necessary. No communication with the database can occur outside of a database transaction (this seems to confuse many developers who are used to the auto-commit mode). Always use clear transaction boundaries, even for read-only operations. Depending on your isolation level and database capabilities this might not be required, but there is no downside if you always demarcate transactions explicitly. Certainly, a single database transaction is going to perform better than many small transactions, even for reading data.

**Atomicity**: Is defined as either all operations can be done or all operation can be undone

**Consistency**: After a transaction is completed successfully, the data in the datastore should be a reliable data. This reliable data is also called as consistent data

**Isolation**: If two transactions are going on the same data then one transaction will not disturb the other transaction

**Durability**: After a transaction is completed, the data in the datastore will be permanent until another transaction is going to be performed on that data

In Hibernate framework, we have Transaction interface that defines the unit of work. It maintains the abstraction from the transaction implementation (JTA, JDBC). A Transaction is associated with Hibernate Session and instantiated by calling the sessionObj.beginTransaction(). The methods of Transaction interface are as follows:

Name Description Syntax

1. begin()

It starts a new transaction.

public void begin() throws HibernateException

1. commit()

It ends the transaction and flushes the associated session.

public void commit() throws HibernateException

1. rollback()

It rolls back the current transaction.

public void rollback()throws HibernateException

1. setTimeout(int seconds) I

it set the transaction timeout for any transaction started by a subsequent call to begin() on this instance.

public void setTimeout(int seconds) throws HibernateException

1. isActive()

It checks if this transaction is still active or not.

public boolean isActive()throws HibernateException

1. wasRolledBack()

It checks if this transaction roll backed successfully or not.

public boolean wasRolledBack()throws HibernateException

1. wasCommitted()

It checks if this transaction committed successfully or not.

public boolean wasCommitted()throws HibernateException

1. registerSynchronization(Synchronization synchronization)

It registers a user synchronization callback for this transaction.

public boolean registerSynchronization(Synchronization synchronization)throws HibernateException

#### Interceptor and Listeners in Hiberanate

#### What is @Transactional Annotation will do

Using "Spring Hibernate Transaction" means that you grant the open/close authority of a transaction for Spring, and you don't need to care about handling it.

This is a typical code part. When you use Hibernate, you have to start 1 transaction and close it after you finish it..

Session session = sessionFactory.getCurrentSession();

try {

session.getTransaction().begin();

// Todo something here

session.getTransaction().commit();

} catch (Exception e) {

e.printStackTrace();

session.getTransaction().rollback();

}

The above manipulation is manual, and it has difficulties in handling professional situations on various methods. Suppose A person send money to B person. You have write two methods. The first method is to debit the A person's account. The second method is to credit the B person's account. These two methods must belong to a Transaction.

Spring AOP allows the configuration to help you manage the Transaction automatically.

you shouldn't make DAO methods transactional, but service methods.

Second, using

@Transactional is a way to let Spring start and commit/rollback transactions for you. So you shouldn't start and commit transactions yourself.

Third: this will only work if you use a transaction manager that knows how to associate a Hibernate session with the transaction (typically, a HibernateTransactionManager). The session factory should also be handled by Spring, and injected by Spring in your DAOs. The code of the DAO should look like this:

Fourth: you should not open a new session, but get the current one, associated to the current transaction by Spring.

JPA and Transaction Management

It's important to notice that JPA on itself does not provide any type of declarative transaction management. When using JPA outside of a dependency injection container, transactions need to be handled programatically by the developer:

UserTransaction utx = entityManager.getTransaction();

try {

utx.begin();

businessLogic();

utx.commit();

} catch(Exception ex) {

utx.rollback();

throw ex;

}

This way of managing transactions makes the scope of the transaction very clear in the code, but it has several disavantages:

it's repetitive and error prone

any error can have a very high impact

errors are hard to debug and reproduce

this decreases the readability of the code base

What if this method calls another transactional method?

Using Spring @Transactional

With Spring @Transactional, the above code gets reduced to simply this:

@Transactional

public void businessLogic() {

... use entity manager inside a transaction ...

}

This is much more convenient and readable, and is currently the recommended way to handle transactions in Spring.

By using @Transactional, many important aspects such as transaction propagation are handled automatically. In this case if another transactional method is called by businessLogic(), that method will have the option of joining the ongoing transaction.

One potential downside is that this powerful mechanism hides what is going on under the hood, making it hard to debug when things don't work.

What does @Transactional mean?

One of the key points about @Transactional is that there are two separate concepts to consider, each with it's own scope and life cycle:

the persistence context

the database transaction

The transactional annotation itself defines the scope of a single database transaction. The database transaction happens inside the scope of apersistence context.

The persistence context is in JPA the EntityManager, implemented internally using an Hibernate Session (when using Hibernate as the persistence provider).

The persistence context is just a synchronizer object that tracks the state of a limited set of Java objects and makes sure that changes on those objects are eventually persisted back into the database.

This is a very different notion than the one of a database transaction. One Entity Manager can be used across several database transactions, and it actually often is.

When does an EntityManager span multiple database transactions?

The most frequent case is when the application is using the Open Session In View pattern to deal with lazy initialization exceptions, see this previous blog post for it's pros and cons.

In such case the queries that run in the view layer are in separate database transactions than the one used for the business logic, but they are made via the same entity manager.

Another case is when the persistence context is marked by the developer as PersistenceContextType.EXTENDED, which means that it can survive multiple requests.

What defines the EntityManager vs Transaction relation?

This is actually a choice of the application developer, but the most frequent way to use the JPA Entity Manager is with the

"Entity Manager per application transaction" pattern. This is the most common way to inject an entity manager:

@PersistenceContext

private EntityManager em;

Here we are by default in "Entity Manager per transaction" mode. In this mode, if we use this Entity Manager inside a @Transactional method, then the method will run in a single database transaction.

How does @PersistenceContext work?

One question that comes to mind is, how can @PersistenceContext inject an entity manager only once at container startup time, given that entity managers are so short lived, and that there are usually multiple per request.

The answer is that it can't: EntityManager is an interface, and what gets injected in the spring bean is not the entity manager itself but a context aware proxy that will delegate to a concrete entity manager at runtime.

Usually the concrete class used for the proxy is

SharedEntityManagerInvocationHandler, this can be confirmed with the help a debugger.

How does @Transactional work then?

The persistence context proxy that implements EntityManager is not the only component needed for making declarative transaction management work. Actually three separate components are needed:

The EntityManager Proxy itself

The Transactional Aspect

The Transaction Manager

Let's go over each one and see how they interact.

The Transactional Aspect

The Transactional Aspect is an 'around' aspect that gets called both before and after the annotated business method. The concrete class for implementing the aspect is TransactionInterceptor.

The Transactional Aspect has two main responsibilities:

At the 'before' moment, the aspect provides a hook point for determining if the business method about to be called should run in the scope of an ongoing database transaction, or if a new separate transaction should be started.

At the 'after' moment, the aspect needs to decide if the transaction should be committed, rolled back or left running.

At the 'before' moment the Transactional Aspect itself does not contain any decision logic, the decision to start a new transaction if needed is delegated to the Transaction Manager.

The Transaction Manager

The transaction manager needs to provide an answer to two questions:

should a new Entity Manager be created?

should a new database transaction be started?

This needs to be decided at the moment the Transactional Aspect 'before' logic is called. The transaction manager will decide based on:

the fact that one transaction is already ongoing or not

the propagation attribute of the transactional method (for example REQUIRES\_NEW always starts a new transaction)

If the transaction manager decides to create a new transaction, then it will:

create a new entity manager

bind the entity manager to the current thread

grab a connection from the DB connection pool

bind the connection to the current thread

The entity manager and the connection are both bound to the current thread using ThreadLocal variables.

They are stored in the thread while the transaction is running, and it's up to the Transaction Manager to clean them up when no longer needed.

Any parts of the program that need the current entity manager or connection can retrieve them from the thread. One program component that does exactly that is the EntityManager proxy.

The EntityManager proxy

The EntityManager proxy (that we have introduced before) is the last piece of the puzzle. When the business method calls for example

entityManager.persist(), this call is not invoking the entity manager directly.

Instead the business method calls the proxy, which retrieves the current entity manager from the thread, where the Transaction Manager put it.

Knowing now what are the moving parts of the @Transactionalmechanism, let's go over the usual Spring configuration needed to make this work.

Putting It All Together

Let's go over how to setup the three components needed to make the transactional annotation work correctly. We start by defining the entity manager factory.

This will allow the injection of Entity Manager proxies via the persistence context annotation:

@Configuration

public class EntityManagerFactoriesConfiguration {

@Autowired

private DataSource dataSource;

@Bean(name = "entityManagerFactory")

public LocalContainerEntityManagerFactoryBean emf() {

LocalContainerEntityManagerFactoryBean emf = ...

emf.setDataSource(dataSource);

emf.setPackagesToScan(new String[] {"your.package"});

emf.setJpaVendorAdapter(

new HibernateJpaVendorAdapter());

return emf;

}

}

The next step is to configure the Transaction Manager and to apply the Transactional Aspect in @Transactional annotated classes:

@Configuration

@EnableTransactionManagement

public class TransactionManagersConfig {

@Autowired

EntityManagerFactory emf;

@Autowired

private DataSource dataSource;

@Bean(name = "transactionManager")

public PlatformTransactionManager transactionManager() {

JpaTransactionManager tm = new JpaTransactionManager();

tm.setEntityManagerFactory(emf);

tm.setDataSource(dataSource);

return tm;

}

}

The annotation @EnableTransactionManagement tells Spring that classes with the @Transactional annotation should be wrapped with the Transactional Aspect. With this the @Transactional is now ready to be used.

Conclusion

The Spring declarative transaction management mechanism is very powerful, but it can be misused or wrongly configured easily.

Understanding how it works internally is helpful when troubleshooting situations when the mechanism is not at all working or is working in an unexpected way.

The most important thing to bear in mind is that there are really two concepts to take into account: the database transaction and the persistence context, each with it's own not readily apparent life cycle.

A future post will go over the most frequent pitfalls of the transactional annotation and how to avoid them.

#### What is propagation ? explain about them

While dealing with Spring managed transactions the developer is able to specify how the transactions should behave in terms of propagation. In other words the developer has the ability to decide how the business methods should be encapsulated in both logical or physical transactions. Methods from distinct Spring beans may be executed in the same transaction scope or actually being spanned across multiple nested transactions. This may lead to details like how does the inner transaction outcome result affects the outer transactions. We will see the different propagation behaviors provided by Spring in the next sections.

This tutorial will only focus in transaction propagation behavior. For other details about Spring transactions you may refer to other tutorials on this website or the official Spring documentation at SpringSource website.

The full source code used in this tutorial is available for download at the bottom of this page. We will only show the relevant parts for a good understanding of the distinct transaction propagation behaviors in Spring.

The full source code uses Hibernate to implement the persistence layer (Spring with Hibernate persistence and transactions example).

REQUIRED behavior

Spring REQUIRED behavior means that the same transaction will be used if there is an already opened transaction in the current bean method execution context. If there is no existing transaction the Spring container will create a new one. If multiple methods configured as REQUIRED behavior are called in a nested way they will be assigned distinct logical transactions but they will all share the same physical transaction. In short this means that if an inner method causes a transaction to rollback, the outer method will fail to commit and will also rollback the transaction. Let's see an example:

Outer bean

@Autowired

private TestDAO testDAO;

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequired();

} catch(RuntimeException e){

// handle exception

}

}

Inner bean

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired() {

throw new RuntimeException("Rollback this transaction!");

}

Note that the inner method throws a RuntimeException and is annotated with REQUIRED behavior. This means that it will use the same transaction as the outer bean, so the outer transaction will fail to commit and will also rollback.

Note: The only exceptions that set a transaction to rollback state by default are the unchecked exceptions (like RuntimeException). If you want checked exceptions to also set transactions to rollback you must configure them to do so, but this will not be covered in this tutorial.

Note 2: When using declarative transactions, ie by using only annotations, and calling methods from the same bean directly (self-invocation), the @Transactional annotation will be ignored by the container. If you want to enable transaction management in self-invocations you must configure the transactions using aspectj, but this will not be covered in this tutorial.

REQUIRES\_NEW behavior

REQUIRES\_NEW behavior means that a new physical transaction will always be created by the container. In other words the inner transaction may commit or rollback independently of the outer transaction, i.e. the outer transaction will not be affected by the inner transaction result: they will run in distinct physical transactions.

Outer bean

@Autowired

private TestDAO testDAO;

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequiresNew(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequiresNew();

} catch(RuntimeException e){

// handle exception

}

}

Inner bean

@Override

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void testRequiresNew() {

throw new RuntimeException("Rollback this transaction!");

}

The inner method is annotated with REQUIRES\_NEW and throws a RuntimeException so it will set its transaction to rollback but will not affect the outer transaction. The outer transaction is paused when the inner transaction starts and then resumes after the inner transaction is concluded. They run independently of each other so the outer transaction may commit successfully.

NESTED behavior

The NESTED behavior makes nested Spring transactions to use the same physical transaction but sets savepoints between nested invocations so inner transactions may also rollback independently of outer transactions. This may be familiar to JDBC aware developers as the savepoints are achieved with JDBC savepoints, so this behavior should only be used with Spring JDBC managed transactions (Spring JDBC transactions example).

MANDATORY behavior

The MANDATORY behavior states that an existing opened transaction must already exist. If not an exception will be thrown by the container.

NEVER behavior

The NEVER behavior states that an existing opened transaction must not already exist. If a transaction exists an exception will be thrown by the container.

NOT\_SUPPORTED behavior

The NOT\_SUPPORTED behavior will execute outside of the scope of any transaction. If an opened transaction already exists it will be paused.

SUPPORTS behavior

The SUPPORTS behavior will execute in the scope of a transaction if an opened transaction already exists. If there isn't an already opened transaction the method will execute anyway but in a non-transactional way.

Quick note about the full source code

The full source code available at the end of this page considers the following MySQL table:

MySQL table used in full source code

CREATE TABLE USER (

ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

USERNAME VARCHAR (32) NOT NULL,

NAME VARCHAR (64) NOT NULL,

UNIQUE (USERNAME)

);

#### What is isolation ? tell all isolation levels ?

Transaction isolation level is a concept that is not exclusive to the Spring framework. It is applied to transactions in general and is directly related with the ACID transaction properties. Isolation level defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and also how and when that changed data becomes available to other transactions. When we define a transaction using the Spring framework we are also able to configure in which isolation level that same transaction will be executed.

Usage example

Using the @Transactional annotation we can define the isolation level of a Spring managed bean transactional method. This means that the transaction in which this method is executed will run with that isolation level:

Isolation level in a transactional method

@Autowired

private TestDAO testDAO;

@Transactional(isolation=Isolation.READ\_COMMITTED)

public void someTransactionalMethod(User user) {

// Interact with testDAO

}

We are defining this method to be executed in a transaction which isolation level is READ\_COMMITTED. We will see each isolation level in detail in the next sections.

READ\_UNCOMMITTED

READ\_UNCOMMITTED isolation level states that a transaction may read data that is still uncommitted by other transactions. This constraint is very relaxed in what matters to transactional concurrency but it may lead to some issues like dirty reads. Let's see the following image:

Dirty read

Transaction isolation level dirty read

In this example Transaction A writes a record. Meanwhile Transaction B reads that same record before Transaction A commits. Later Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ\_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.

Note: READ\_UNCOMMITTED is also vulnerable to non-repeatable reads and phantom reads. We will also see these cases in detail in the next sections.

READ\_COMMITTED

READ\_COMMITTED isolation level states that a transaction can't read data that is not yet committed by other transactions. This means that the dirty read is no longer an issue, but even this way other issues may occur. Let's see the following image:

Non-repeatable read

Transaction isolation level repeatable read

In this example Transaction A reads some record. Then Transaction B writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.

Note: READ\_COMMITTED is also vulnerable to phantom reads. We will also see this case in detail in the next section.

REPEATABLE\_READ

REPEATABLE\_READ isolation level states that if a transaction reads one record from the database multiple times the result of all those reading operations must always be the same. This eliminates both the dirty read and the non-repeatable read issues, but even this way other issues may occur. Let's see the following image:

Phantom read

Transaction isolation level phantom read

In this example Transaction A reads a range of records. Meanwhile Transaction B inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

SERIALIZABLE

SERIALIZABLE isolation level is the most restrictive of all isolation levels. Transactions are executed with locking at all levels (read, range and write locking) so they appear as if they were executed in a serialized way. This leads to a scenario where none of the issues mentioned above may occur, but in the other way we don't allow transaction concurrency and consequently introduce a performance penalty.

DEFAULT

DEFAULT isolation level, as the name states, uses the default isolation level of the datastore we are actually connecting from our application.

To summarize, the existing relationship between isolation level and read phenomena may be expressed in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **dirty reads** | **non-repeatable reads** | **phantom reads** |
| **READ\_UNCOMMITTED** | yes | yes | yes |
| **READ\_COMMITTED** | no | yes | yes |
| **REPEATABLE\_READ** | no | no | yes |
| **SERIALIZABLE** | no | no | no |

#### What is cascade ? what all value in cascade explain all of them

##### cascade="none",

the default, tells Hibernate to ignore the association.

##### cascade="save-update"

tells Hibernate to navigate the association when the

transaction is committed and when an object is passed to save() or

update() and save newly instantiated transient instances and persist changes to

detached instances.

##### cascade="delete"

tells Hibernate to navigate the association and delete persistent

instances when an object is passed to delete().

##### cascade="all"

means to cascade both save-update and delete, as well as

calls to evict and lock.

##### cascade="all-delete-orphan"

means the same as cascade="all" but, in addition,

Hibernate deletes any persistent entity instance that has been removed

(dereferenced) from the association (for example, from a collection).

6) cascade="delete-orphan" Hibernate will delete any persistent entity

instance that has been removed (dereferenced) from the association (for

example, from a collection).

#### Hibernate cache ?

#### How can we achieve relationship in hibernate and the types of it.

#### What is Cache in Hibernate ?What is session cache and session factory cache.

#### what make a sense to use hibernate as DAO technologies in your project?

#### what is dialect? And where we need to configure Dialect?

#### What is HQL?

#### what is Session level cache?

#### what is application level cache?

#### when to use application level cache?

#### advantages of hibernate over jpa

#### how many tables needed in many to many and why.

#### Many to many relationship in hibernate

#### What is Bi-Directional relationship in Hibernate.

#### Write Annotation based Hibernate mapping for OneToMany Relationship.

#### When to use List and when to use Set in Hibernate?

SessionFactory which acts as singleton for the whole application.

### Hibernate functionality/flow/usage can be described as follows:

On Application startup, hibernate reads it configuration file(hibernate.cfg.xml or hibernate.properties) which contains information required to make the connection with underlying database and mapping information. Based on this information, hibernate creates Configuration Object , which in turns creates SessionFactory which acts as singleton for the whole application.

Hibernate creates instances of entity classes.Entity classes are java classes which are mapped to the database table using metadata(XML/Annotaitons).These instances are called transient objects as they are not yet persisted in database.

To persist an object, application ask for a Session from SessionFactory which is a factory for Session.Session represent a physical database connection.

Application then starts the transaction to make the unit of work atomic, & uses Session API’s to finally persist the entity instance in database.Once the entity instance persisted in database, it’s known as persistent object as it represent a row in database table.Application then closes/commits the transaction followed by session close.

Once the session gets closed , the entity instance becomes detatched which means it still contains data but no more attached to the database table & no more under the management of Hibernate. Detatched objects can again become persistent when associated with a new Session, or can be garbage collected once no more used.

### **Below is the brief description of commonly used core API’s in a typical application persistence with Hibernate**.

#### Configuration (org.hibernate.cfg.Configuration)

It allows the application on startup, to specify properties and mapping documents to be used when creating a SessionFactory. Properties file contains database connection setup info while mapping specifies the classes to be mapped.

#### SessionFactory (org.hibernate.SessionFactory)

It’s a thread-safe immutable object created per database & mainly used for creating Sessions.It caches generated SQL statements and other mapping metadata that Hibernate uses at runtime.

We can get instance of org.hibernate.Session using SessionFactory.

#### Session (org.hibernate.Session)

It’s a single-threaded object used to perform create, read, update and delete operations for instances of mapped entity classes. Since it’s not thread-safe, it should not be long-lived and each thread/transaction should obtain its own instance from a SessionFactory.

It wraps JDBC java.sql.Connection and works as a factory for org.hibernate.Transaction.

Persistent objects: Persistent objects are short-lived, single threaded objects that contains persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one org.hibernate.Session.

Transient objects: Transient objects are persistent classes instances that are not currently associated with a org.hibernate.Session. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed org.hibernate.Session.

#### Transaction (org.hibernate.Transaction)

It’s a single-thread object used by the application to define units of work. A transaction is associated with a Session. Transactions abstract application code from underlying transaction implementations(JTA/JDBC), allowing the application to control transaction boundaries via a consistent API. It’s an Optional API and application may choose not to use it.

#### Query (org.hibernate.Query)

A single-thread object used to perform query on underlying database. A Session is a factory for Query. Both HQL(Hibernate Query Language) & SQL can be used with Query object.

#### Criteria (org.hibernate.Criteria)

It is an alternative to HQL , very useful for the search query involving multiple conditions.

Object-relational mapping or ORM is the programming technique to map application domain model objects to the relational database tables. Hibernate is java based ORM tool that provides framework for mapping application domain objects to the relational database tables and vice versa.

Some of the benefits of using Hibernate as ORM tool are:

Hibernate supports mapping of java classes to database tables and vice versa. It provides features to perform CRUD operations across all the major relational databases.

Hibernate eliminates all the boiler-plate code that comes with JDBC and takes care of managing resources, so we can focus on business use cases rather than making sure that database operations are not causing resource leaks.

Hibernate supports transaction management and make sure there is no inconsistent data present in the system.

Since we use XML, property files or annotations for mapping java classes to database tables, it provides an abstraction layer between application and database.

Hibernate helps us in mapping joins, collections, inheritance objects and we can easily visualize how our model classes are representing database tables.

Hibernate provides a powerful query language (HQL) that is similar to SQL. However, HQL is fully object-oriented and understands concepts like inheritance, polymorphism and association.

Hibernate also offers integration with some external modules. For example Hibernate Validator is the reference implementation of Bean Validation (JSR 303).

Hibernate is an open source project from Red Hat Community and used worldwide. This makes it a better choice than others because learning curve is small and there are tons of online documentations and help is easily available in forums.

Hibernate is easy to integrate with other Java EE frameworks, it’s so popular that Spring Framework provides built-in support for integrating hibernate with Spring applications.

#### Hibernate Architecture

