# 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.

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

#### explain hibernate dialect property

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

#### What is transaction ?

#### What is @Transactional Annotation will do

#### What is propagation ? explain about them

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

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

#### 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

